



DEITEL® DEVELOPER SERIES

# Concepts

Functional-Style Programming

Standard Library

Concurrency

# Coroutines

# Modules

Apple Xcode/Clang

Security

Microsoft Visual C++



# for Programmers

## Ranges

Lambdas

Performance

STL/Parallel STL

## Modern C++

Open Source Libraries

Visualization

# Text Formatting

PAUL DEITEL • HARVEY DEITEL

**Deitel® Developer Series**

# C++ 20 for Programmers

**Paul Deitel • Harvey Deitel**





Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 2021 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions](http://www.pearson.com/permissions).

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

ISBN-10: 0-13-690569-1

# Contents

## Preface

### **Part 1: C++ Fundamentals Quickstart**

[Chapter 1: Introduction and Test-Driving a C++ Application](#)

[Chapter 2: Introduction to C++ Programming](#)

[Chapter 3: Control Statements, Part 1; Intro to C++20 Text](#)

[Formatting](#)

[Chapter 4: Control Statements, Part 2](#)

[Chapter 5: Functions; Intro to Functional-Style Programming;](#)

[Intro to C++20 Modules](#)

### **Part 2: Arrays, Pointers, Strings and Files**

[Chapter 6: Class Templates `array` and `vector`; Intro to C++20 Concepts and Ranges](#)

[Chapter 7: Pointers](#)

[Chapter 8: Class `string` and Regular Expressions](#)

[Chapter 9: File Processing and String Stream Processing](#)

### **Part 3: Object-Oriented Programming**

[Chapter 10: Introduction to Classes](#)

[Chapter 11: Classes and Objects: A Deeper Look](#)

[Chapter 12: Inheritance](#)

[Chapter 13: Polymorphism](#)

[Chapter 14: Operator Overloading](#)

[Chapter 15: Exceptions: A Deeper Look](#)

### **Part 4: Standard Library Containers, Iterators and Algorithms**

[Chapter 16: Standard Library Containers and Iterators](#)

[Chapter 17: Standard Library Algorithms](#)

### **Part 5: Other Topics**

[Chapter 18: Intro to Custom Templates](#)

[Chapter 19: Stream I/O and C++20 Text Formatting: A Deeper Look](#)

[Chapter 20: Concurrent Programming; Intro to C++20 Coroutines](#)

[Chapter 21: Bits, Characters, C Strings and `structs`](#)

[Chapter 22: Other Topics; A Look Toward C++23 and Contracts](#)

## **Part 6: Appendices**

[Appendix A: Operator Precedence and Grouping](#)

[Appendix B: Character Set](#)

[Appendix C: Fundamental Types](#)

[Appendix D: Number Systems](#)

[Appendix E: Preprocessor](#)

[Appendix F: C Legacy Code Topics](#)

[Appendix G: Using the Visual Studio Debugger](#)

[Appendix H: Using the GNU C++ Debugger](#)

[Appendix I: Using the Xcode Debugger](#)

# Table of Contents

## Preface

## Part 1: C++ Fundamentals Quickstart

### 1. Introduction and Test-Drives

- 1.1 Introduction
- 1.2 Test-Driving a C++20 Application

### 2. Intro to C++ Programming

- 2.1 Introduction
- 2.2 First Program in C++: Displaying a Line of Text
- 2.3 Modifying Our First C++ Program
- 2.4 Another C++ Program: Adding Integers
- 2.5 Arithmetic
- 2.6 Decision Making: Equality and Relational Operators
- 2.7 Objects Natural: Creating and Using Objects of Standard Library Class `string`
- 2.8 Wrap-Up

### 3. Control Statements: Part 1

- 3.1 Introduction
- 3.2 Control Structures
- 3.3 `if` Single-Selection Statement
- 3.4 `if...else` Double-Selection Statement
- 3.5 `while` Iteration Statement
- 3.6 Counter-Controlled Iteration
- 3.7 Sentinel-Controlled Iteration
- 3.8 Nested Control Statements
- 3.9 Compound Assignment Operators
- 3.10 Increment and Decrement Operators
- 3.11 Fundamental Types Are Not Portable
- 3.12 Objects Natural Case Study: Arbitrary Sized Integers

[3.13 C++20 Feature Mock-Up—Text Formatting with  
Function `format`](#)

[3.14 Wrap-Up](#)

## **4. Control Statements, Part 2**

[4.1 Introduction](#)

[4.2 Essentials of Counter-Controlled Iteration](#)

[4.3 `for` Iteration Statement](#)

[4.4 Examples Using the `for` Statement](#)

[4.5 Application: Summing Even Integers](#)

[4.6 Application: Compound-Interest Calculations](#)

[4.7 `do...while` Iteration Statement](#)

[4.8 `switch` Multiple-Selection Statement](#)

[4.9 C++17: Selection Statements with Initializers](#)

[4.10 `break` and `continue` Statements](#)

[4.11 Logical Operators](#)

[4.12 Confusing the Equality \(==\) and Assignment \(=\)](#)

[Operators](#)

[4.13 C++20 Feature Mock-Up: `\[\[likely\]\]` and  
`\[\[unlikely\]\]` Attributes](#)

[4.14 Objects Natural Case Study: Using the `miniz-cpp`  
Library to Write and Read ZIP files](#)

[4.15 C++20 Feature Mock-Up: Text Formatting with Field  
Widths and Precisions](#)

[4.16 Wrap-Up](#)

## **5. Functions**

[5.1 Introduction](#)

[5.2 Program Components in C++](#)

[5.3 Math Library Functions](#)

[5.4 Function Definitions and Function Prototypes](#)

[5.5 Order of Evaluation of a Function's Arguments](#)

[5.6 Function-Prototype and Argument-Coercion Notes](#)

[5.7 C++ Standard Library Headers](#)

[5.8 Case Study: Random-Number Generation](#)

[5.9 Case Study: Game of Chance; Introducing Scoped  
enums](#)

- [5.10 C++11's More Secure Nondeterministic Random Numbers](#)
- [5.11 Scope Rules](#)
- [5.12 Inline Functions](#)
- [5.13 References and Reference Parameters](#)
- [5.14 Default Arguments](#)
- [5.15 Unary Scope Resolution Operator](#)
- [5.16 Function Overloading](#)
- [5.17 Function Templates](#)
- [5.18 Recursion](#)
- [5.19 Example Using Recursion: Fibonacci Series](#)
- [5.20 Recursion vs. Iteration](#)
- [5.21 C++17 and C++20: `\[\[nodiscard\]\]` Attribute](#)
- [5.22 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz  
Xndmwwgħliz](#)
- [5.23 Wrap-Up](#)

## **Part 2: Arrays, Pointers, Strings and Files**

[Chapter 6: Class Templates `array` and `vector`; Intro to C++20 Concepts and Ranges](#)

[Chapter 7: Pointers](#)

[Chapter 8: Class `string` and Regular Expressions](#)

[Chapter 9: File Processing and String Stream Processing](#)

## **Part 3: Object-Oriented Programming**

[Chapter 10: Introduction to Classes](#)

[Chapter 11: Classes and Objects: A Deeper Look](#)

[Chapter 12: Inheritance](#)

[Chapter 13: Polymorphism](#)

[Chapter 14: Operator Overloading](#)

[Chapter 15: Exceptions: A Deeper Look](#)

## **Part 4: Standard Library Containers, Iterators and Algorithms**

[Chapter 16: Standard Library Containers and Iterators](#)

[Chapter 17: Standard Library Algorithms](#)

## **Part 5: Other Topics**

[Chapter 18: Intro to Custom Templates](#)

[Chapter 19: Stream I/O and C++20 Text Formatting: A Deeper Look](#)

[Chapter 20: Concurrent Programming; Intro to C++20 Coroutines](#)

[Chapter 21: Bits, Characters, C Strings and `structs`](#)

[Chapter 22: Other Topics; A Look Toward C++23 and Contracts](#)

## **Part 6: Appendices**

[Appendix A: Operator Precedence and Grouping](#)

[Appendix B: Character Set](#)

[Appendix C: Fundamental Types](#)

[Appendix D: Number Systems](#)

[Appendix E: Preprocessor](#)

[Appendix F: C Legacy Code Topics](#)

[Appendix G: Using the Visual Studio Debugger](#)

[Appendix H: Using the GNU C++ Debugger](#)

[Appendix I: Using the Xcode Debugger](#)

# Preface

Welcome to the C++ programming language and *C++20 for Programmers*. This book presents leading-edge computing technologies for software developers.

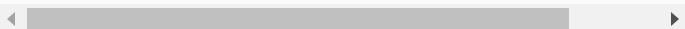
These are exciting times in the programming-languages community with each of the major languages striving to keep pace with compelling new programming technologies. The ISO C++ Standards Committee now releases a new standard every three years and the compiler vendors implement the new features promptly. *C++20 for Programmers* is based on the new C++20 standard.

## Live-Code Approach

At the heart of the book is the Deitel signature live-code approach. We present most concepts in the context of complete working programs followed by one or more sample executions. 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 hundreds of code examples. All the source code is available at

---

<https://www.deitel.com/c-plus-plus-20-for-programmers>



We recommend that you compile and run each program as you study it.

## “Rough-Cut” E-Book for O'Reilly Online Learning Subscribers

You are viewing an early-access “rough cut” of *C++20 for Programmers*. **We prepared this content carefully, but it has not yet been reviewed or copy edited and is subject to change.** As we complete each chapter, we'll post it here.

Please send any corrections, comments, questions and

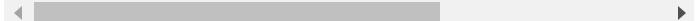
suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I'll respond promptly. Check here frequently for updates.

## “Sneak Peek” Videos for O'Reilly Online Learning Subscribers

As an O'Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

---

<https://learning.oreilly.com/videos/c-20-fundamentals>



Co-author Paul Deitel immediately records each video lesson as we complete each rough-cut e-book chapter. Lessons go live on O'Reilly Online Learning a few days later. Again, check here frequently for updates.

## Modern C++

**20** The C++ programming language is popular for developing systems software, embedded systems, operating systems, real-time systems, computer games, communications systems and other high-performance computer applications. *C++20 for Programmers* is an introductory-through-intermediate-level, professional tutorial presentation of C++. The book is not a full-language reference manual, nor is it a “cookbook.” We focus on Modern C++, which consists of the four most recent C++ standards—C++11, C++14, C++17 and C++20. Throughout the book, features from these standards are marked with icons in the margins like the 20 icon shown here.

## Compilers

We tested every program in the book on three popular free compilers:

- Visual C++ in Microsoft Visual Studio Community edition on Windows,
- Clang in Xcode on macOS, and

- GNU C++ on Linux and in the GNU Compiler Collection (GCC) Docker container.

At the time of this writing, some C++20 features are fully implemented in all three compilers, some are implemented in a subset of the three and some are not implemented at all. We point out these issues as appropriate and will update our online content as the compiler vendors implement the rest of C++20's features. C++20 compiler support for many more compilers is tracked at

---

[https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support)



We'll also post updates on the book's website at

---

<https://deitel.com/books/c-plus-plus-20-for-programmers>



## Target Audiences

*C++20 for Programmers* and our *C++20 Fundamentals*

*LiveLessons* videos have several target audiences:

- Non-C++ software developers who are about to do a C++ project and want to learn the latest version of the language, C++20, in the context of a professional-style, language tutorial.
- Software developers who may have learned C++ in college or used it professionally some time ago and want to refresh their C++ knowledge in the context of C++20.
- C++ software developers who want to learn C++20 in the context of a professional-style, language tutorial.

## **PERF SECURITY Focus on Performance and Security Issues**

Throughout the book, we call your attention to performance and security issues with the icons you see here in the margin.

## “Objects Natural” Learning Approach

In your C++ programs, you'll create and use many objects of carefully-developed-and-tested preexisting classes that enable you to perform significant tasks with minimal code. These classes typically come from:

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

To help you appreciate this style of programming, early in the book you'll create and use objects of preexisting classes before creating your own custom classes in later chapters. We call this the “objects natural” approach.

Our “Objects Natural” Learning Approach evolved organically as we worked on our *Python for Programmers* book (<https://learning.oreilly.com/library/view/python-for-programmers/9780135231364/>) and our *Python Fundamentals LiveLessons* videos (<https://learning.oreilly.com/videos/python-fundamentals/9780135917411>), but *C++20 for Programmers* is the first book in which we're using the term, “Objects Natural.”

## **“Rough-Cut” Table of Contents (Subject to Change)**

### **Part 1: C++ Fundamentals Quickstart**

1. Introduction and Test-Driving a C++ Application
2. Introduction to C++ Programming
3. Control Statements, Part 1; Intro to C++20 Text Formatting
4. Control Statements, Part 2
5. Functions; Intro to Functional-Style Programming; Intro to C++20 Modules

## **Part 2: Arrays, Pointers, Strings and Files**

6. Class Templates `array` and `vector`; Intro to C++20

Concepts and Ranges

7. Pointers

8. Class `string` and Regular Expressions

9. File Processing and String Stream Processing

## **Part 3: Object-Oriented Programming**

10. Introduction to Classes

11. Classes and Objects: A Deeper Look

12. Inheritance

13. Polymorphism

14. Operator Overloading

15. Exceptions: A Deeper Look

## **Part 4: Standard Library Containers, Iterators and Algorithms**

16. Standard Library Containers and Iterators

17. Standard Library Algorithms

## **Part 5: Other Topics**

18. Intro to Custom Templates

19. Stream I/O and C++20 Text Formatting: A Deeper Look

20. Concurrent Programming; Intro to C++20 Coroutines

21. Bits, Characters, C Strings and `structs`

22. Other Topics; A Look Toward C++23 and Contracts

## **Part 6: Appendices**

A. Operator Precedence and Grouping

B. Character Set

C. Fundamental Types

D. Number Systems

E. Preprocessor

F. C Legacy Code Topics

[G. Using the Visual Studio Debugger](#)

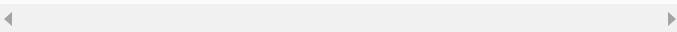
[H. Using the GNU C++ Debugger](#)

[I. Using the Xcode Debugger](#)

## Contacting the Authors

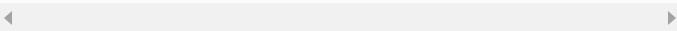
As you read the book, if you have questions, we're easy to reach at

[deitel@deitel.com](mailto:deitel@deitel.com)



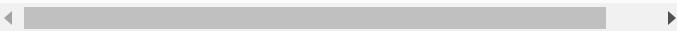
or

<https://deitel.com/contact-us>



We'll respond promptly. For book updates, visit

<https://deitel.com/c-plus-plus-20-for-programmer>



## Join the Deitel & Associates, Inc. Social Media Communities

Join the Deitel social media communities on

- Facebook —

<https://facebook.com/DeitelFan>

- LinkedIn® —

<https://bit.ly/DeitelLinkedIn>

- Twitter® —

<https://twitter.com/deitel>

- YouTube® —

<https://youtube.com/DeitelTV>

## About the Authors

**Paul J. Deitel**, CEO and Chief Technical Officer of Deitel & Associates, Inc., is an MIT graduate with 39 years of experience 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 internationally,

including Pearson Education through O'Reilly Online Learning, 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, Nortel Networks, Puma, iRobot, UCLA and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video/webinar authors.

**Dr. Harvey M. Deitel**, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 59 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. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., 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.

### **About Deitel® & Associates, Inc.**

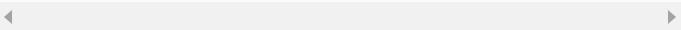
Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, mobile app development and Internet-and-web software technology. The company's training 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 at client sites worldwide on major programming languages.

Through its 45-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in **print** and **e-book** formats, **LiveLessons** video courses, **O'Reilly Online Learning** live webinars and **Revel™** interactive multimedia courses.

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

---

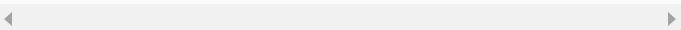
<https://deitel.com/training>



To request a proposal for on-site, instructor-led training worldwide, write to:

---

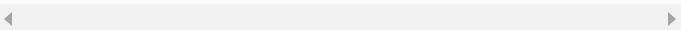
[deitel@deitel.com](mailto:deitel@deitel.com)



Individuals wishing to purchase Deitel books can do so at

---

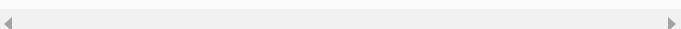
<https://www.amazon.com>



Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

---

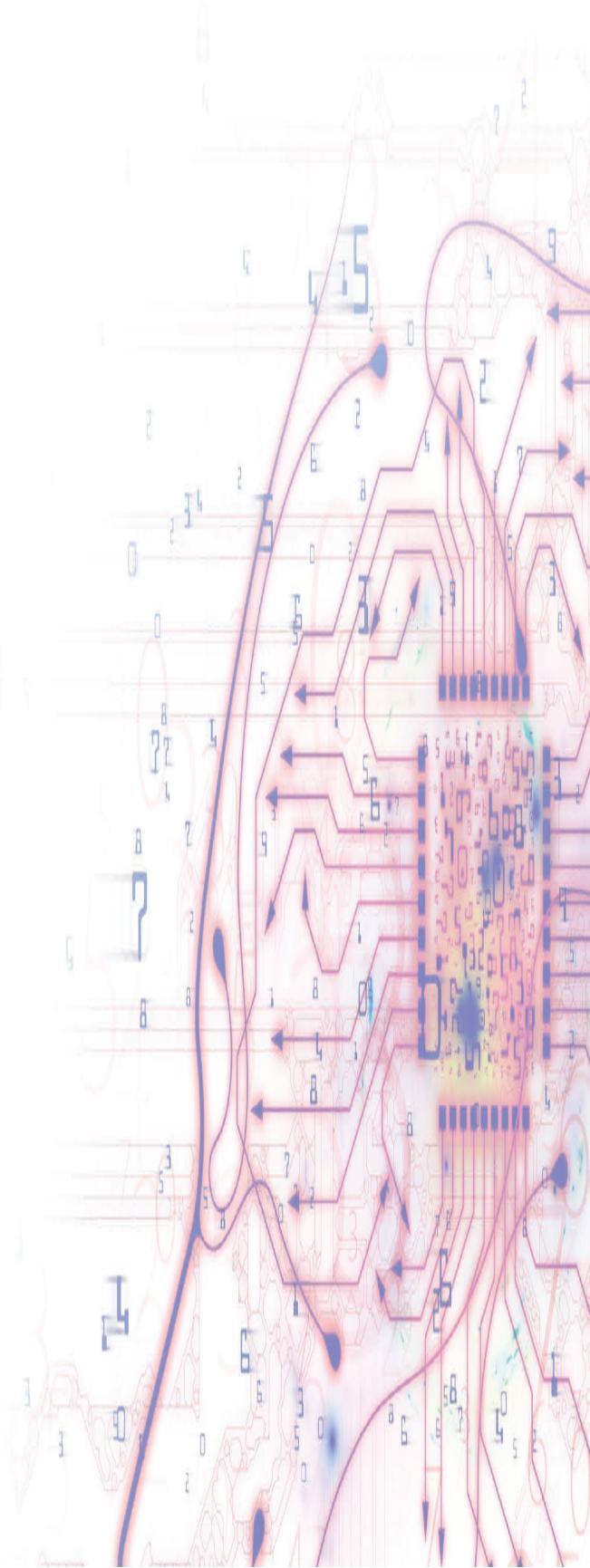
<https://www.informit.com/store/sales.aspx>

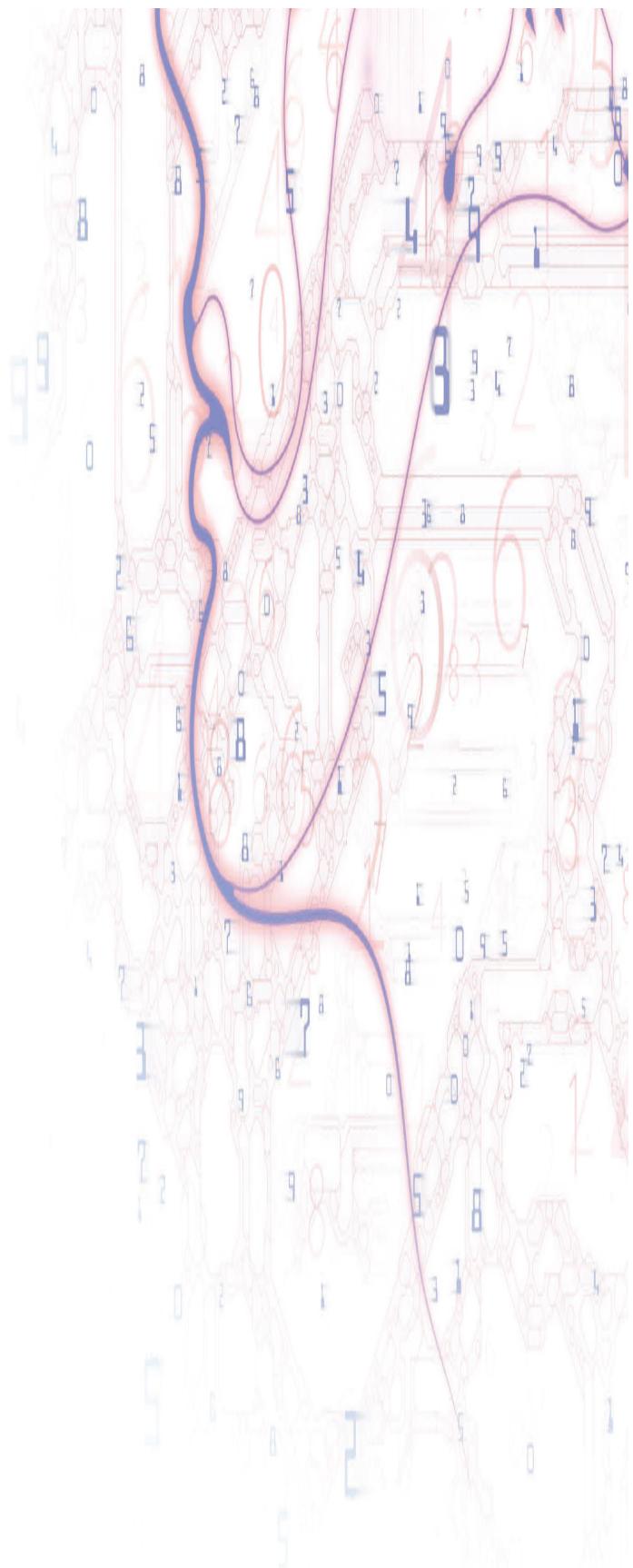


# **Part 1: C++ Fundamentals**

## **Quickstart**

## **1. Introduction and Test-Drives**





## Objectives

In this chapter, you'll do one or more of the following:

- Test-drive a C++20 application in the Visual C++ compiler in Microsoft Visual Studio Community edition on Windows.
- Test-drive a C++20 application in the Clang compiler in Xcode on macOS.
- Test-drive a C++20 application in the GNU C++ compiler on Linux.
- Test-drive a C++20 application in the GNU Compiler Collection (GCC) Docker Container in Docker running natively over Windows 10, macOS and/or Linux.

---

### [1.1 Introduction](#)

### [1.2 Test-Driving a C++20 Application](#)

#### [1.2.1 Compiling and Running a C++20](#)

##### [Application with Visual Studio 2019](#)

##### [Community Edition on Windows 10](#)

#### [1.2.2 Compiling and Running a C++20](#)

##### [Application with Xcode on macOS](#)

#### [1.2.3 Compiling and Running a C++20](#)

##### [Application with GNU C++ on Linux](#)

#### [1.2.4 Compiling and Running a C++20](#)

##### [Application with GNU C++ in the GCC](#)

##### [Docker Container in Docker Running](#)

##### [Natively Over Windows 10, macOS and/or](#)

##### [Linux](#)

---

## 1.1 INTRODUCTION

Welcome to C++—one of the world’s most widely used, high-performance, computer-programming languages—and its current version C++20.

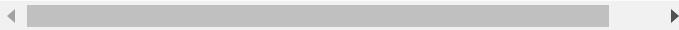
If you’re reading this, you’re on the O’Reilly Online Learning platform (formerly called Safari) viewing an early-access

Rough Cut of our forthcoming book *C++20 for Programmers*, scheduled for publication this summer. **We have prepared this content carefully, but it has not yet been peer reviewed or copy edited and is subject to change.** When we complete this chapter, we'll post the reviewed and copy edited version here.

Please send any corrections, comments, questions and suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I'll respond promptly. Check for updates here and on the book's web page:

---

<https://deitel.com/c-plus-plus-20-for-programmer>



This book is written for developers using one or more of the following popular desktop platforms—Microsoft Windows 10, macOS and Linux. We tested every program on three popular free compilers:

- Visual C++ in Microsoft Visual Studio Community edition on Windows 10,
- Clang in Xcode on macOS, and
- GNU C++ on Linux and in the GNU Compiler Collection (GCC) Docker container.<sup>1</sup>

<sup>1</sup>. At Deitel, we use current, powerful multicore Apple Mac computers that enable us to run macOS natively, and Windows 10 and Linux through virtual machines in VMWare Fusion. Docker runs natively on Windows, macOS and Linux systems.

This early-access version of [Chapter 1](#) contains test-drives demonstrating how to compile and run a C++20 application using these compilers and platforms. The published version of this chapter will contain additional introductory material.

At the time of this writing, some C++20 features are fully implemented in all three compilers, some are implemented in a subset of the three and some are not implemented at all. We point out these issues as appropriate and will update our online content as the compiler vendors implement the rest of C++20's features. C++20 compiler support for many more compilers is tracked at:

---

[https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support)

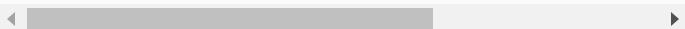


## “Sneak Peek” Videos for O’Reilly Online Learning Subscribers

As an O’Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

---

<https://learning.oreilly.com/videos/c-20-fundamentals>



Co-author Paul Deitel immediately records each video lesson as we complete the corresponding chapter. Lessons go live on O’Reilly Online Learning a few days later. Again, check here frequently for updates.

## 1.2 TEST-DRIVING A C++20 APPLICATION

In this section, you’ll compile, run and interact with your first C++ application—a guess-the-number game, which picks a random number from 1 to 1000 and prompts you to guess it. If you guess correctly, the game ends. If you guess incorrectly, the application indicates whether your guess is higher or lower than the correct number. There’s no limit on the number of guesses you can make.

Usually, this application randomly selects the correct answer as you execute the program. We’ve disabled that aspect of the application so that it uses the same correct answer every time the program executes (though this may vary by compiler). So, you can use the same guesses we use and see the same results.

### Summary of the Test-Drives

We’ll demonstrate running a C++ application using:

- Microsoft Visual Studio 2019 Community edition for Windows (Section 1.2.1)
- Clang in Xcode on macOS (Section 1.2.2).
- GNU C++ in a shell on Linux (Section 1.2.3)

- GNU C++ in a shell running inside the GNU Compiler Collection (GCC) Docker container. This requires Docker to be installed and running.

You need to read only the section that corresponds to your platform. At the time of this writing, GNU C++ supports the most C++20 features of the three compilers we use.

### **1.2.1 Compiling and Running a C++20 Application with Visual Studio 2019 Community Edition on Windows 10**

In this section, you'll run a C++ program on Windows using Microsoft Visual Studio 2019 Community edition. There are several versions of Visual Studio available—on some versions, the options, menus and instructions we present might differ slightly. From this point forward, we'll simply say “Visual Studio” or “the IDE.”

#### **Step 1: Checking Your Setup**

If you have not already done so, read the Before You Begin section of this book for instructions on installing the IDE and downloading the book's code examples.

#### **Step 2: Launching Visual Studio**

Open Visual Studio from the **Start** menu. The IDE displays the following **Visual Studio 2019** window containing:

# Visual Studio 2019

Open recent

Get started

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Clone or check out code  
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution  
Open a local Visual Studio project or .sln file

Open a local folder  
Navigate and edit code within any folder

Create a new project  
Choose a project template with code scaffolding to get started

Continue without code →

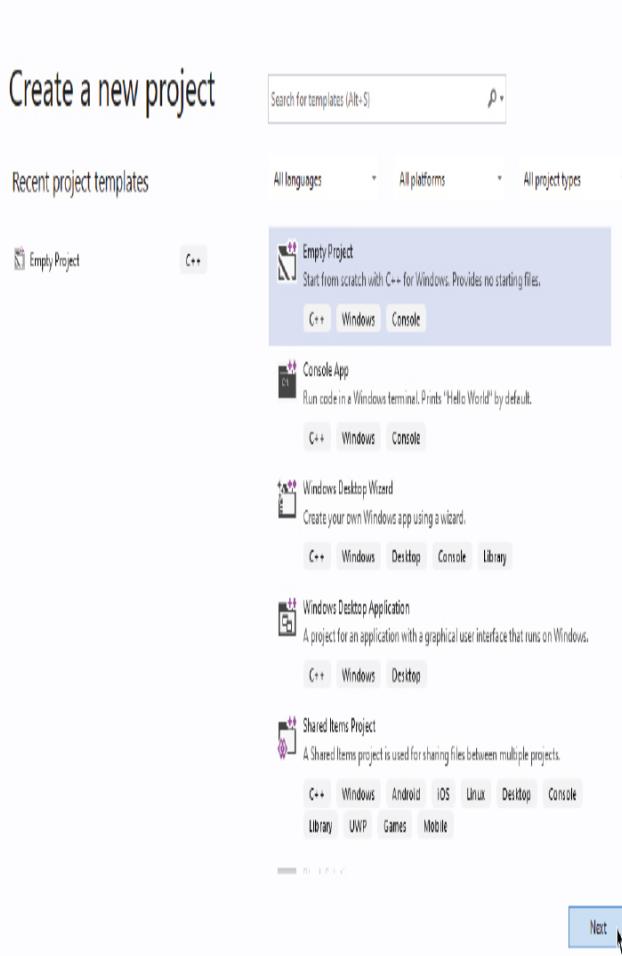
Close this window for now by clicking the X in its tab—you can access this window any time by selecting **File > Start Window**. We use the > character to indicate selecting a menu item from a menu. For example, the notation **File > Open** indicates that you should select the **Open** menu item from the **File** menu.

## Step 3: Creating a Project

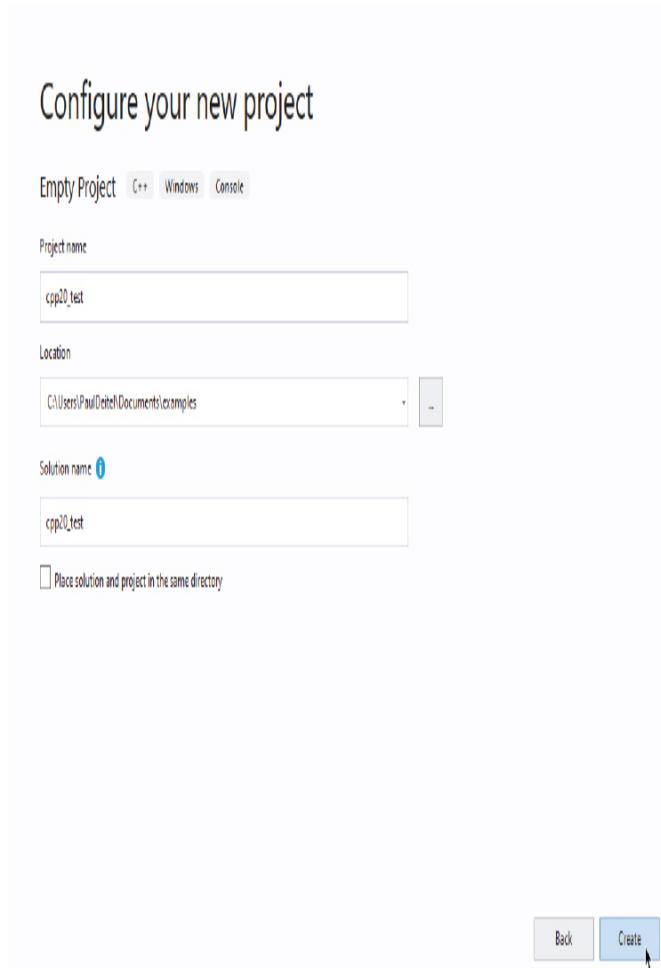
A **project** is a group of related files, such as the C++ source-code files that compose an application. Visual Studio organizes applications into projects and **solutions**, which contain one or more projects. Multiple-project solutions are used to create large-scale applications. Each application in this book requires only a single-project solution.

To work our code examples, you'll begin with an **Empty Project** and add files to it. To create a project:

- 1.** Select **File > New > Project...** to display the **Create a New Project** dialog:



- 2.** In the preceding dialog, select the **Empty Project** template with the tags **C++**, **Windows** and **Console**. This template is for programs that execute at the command line in a Command Prompt window. Depending on the version of Visual Studio you're using and the options you have installed, there may be many other project templates installed. You can narrow down your choices using the **Search for templates** textbox and the drop-down lists below it. Click **Next** to display the **Configure your new project** dialog:



3. Provide a **Project name** and **Location** for your project.

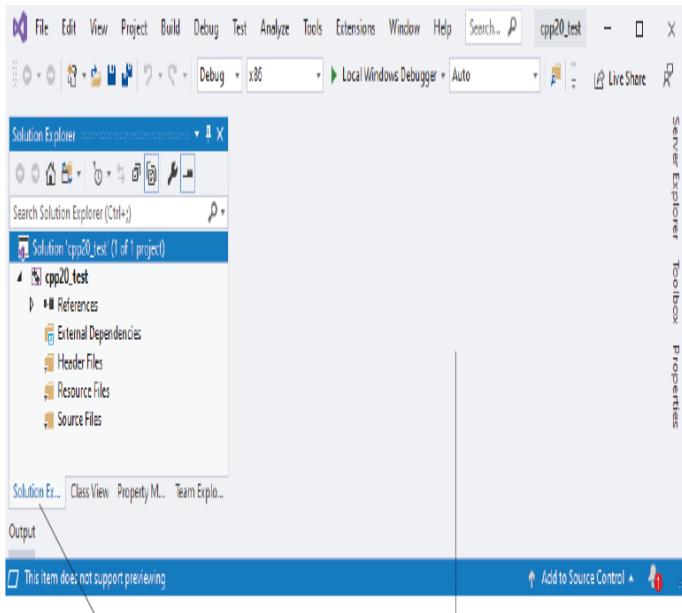
For the **Project name**, we specified `cpp20_test`.

For the **Location**, we selected the `examples` folder containing this book's code examples. Click **Create** to open your new project in Visual Studio.

At this point, the Visual Studio creates your project, places its folder in

`C:\Users\YourUserAccount\Documents\examples`

(or the folder you specified) and opens the main window:



Solution Explorer shows the solution's contents

Source-code editors appear as tabbed windows here

This window displays editors as tabbed windows (one for each file) when you're editing code. On the left side is the **Solution Explorer** for viewing and managing your application's files. In this book's examples, you'll typically place each program's code files in the **Source Files** folder. If the **Solution Explorer** is not displayed, you can display it by selecting **View > Solution Explorer**.

#### Step 4: Adding the `GuessNumber.cpp` File into the Project

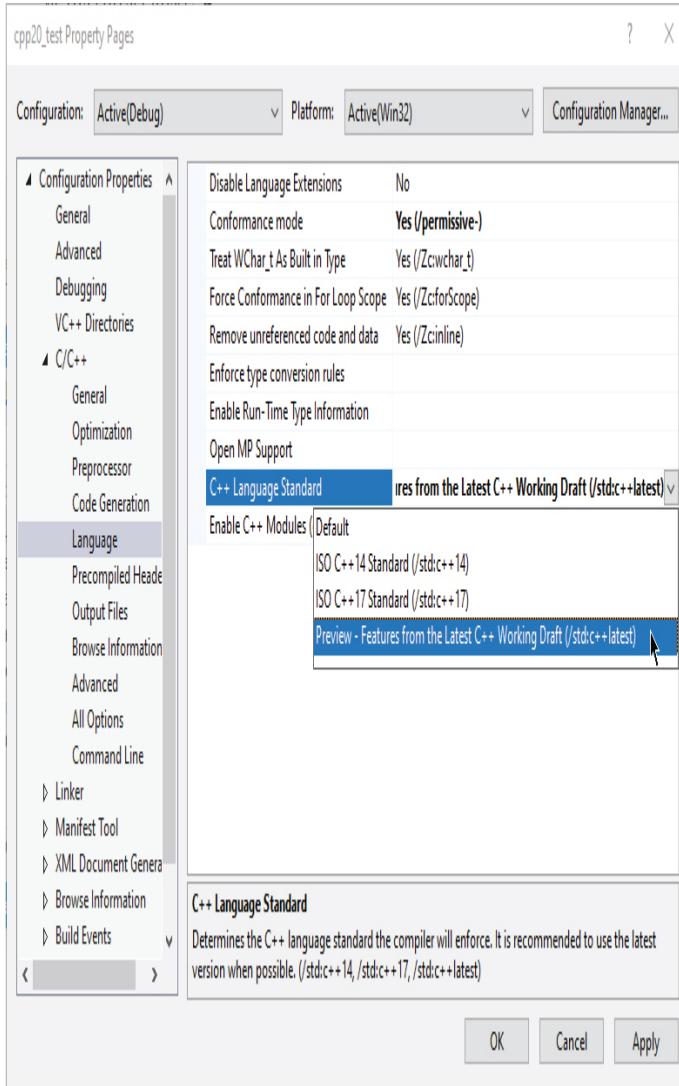
Next, you'll add `GuessNumber.cpp` to the project you created in *Step 3*. In the **Solution Explorer**:

1. Right-click the **Source Files** folder and select **Add > Existing Item....**
  2. In the dialog that appears, navigate to the `ch01` subfolder of the book's `examples` folder, `GuessNumber.cpp` and click **Add**.<sup>2</sup>
2. For the multiple source-code-file programs that you'll see in later chapters, select all the files for a given program. When you begin creating programs yourself, you can right click the **Source Files** folder and select **Add > New Item...** to display a dialog for adding a new file.

#### Step 5: Configuring Your Project to Use C++20

The Visual C++ compiler in Visual Studio supports several versions of the C++ standard. For this book, we use C++20, which we must configure in our project's settings:

1. Right-click the project's node——in the Solution Explorer and select **Properties** to display the project's **cpp20\_test Property Pages** dialog:



2. In the left column, expand the **C/C++** node, then select **Language**.
3. In the right column, click in the field to the right of **C++ Language Standard**, click the down arrow, then select **Preview - Features from the Latest C++ Working Draft (/std:c++latest)** and click **OK**. In a

future version of Visual Studio, Microsoft will change this option to **ISO C++20 Standard (/std:c++20)**.

### Step 6: Compiling and Running the Project

To compile and run the project so you can test-drive the application, select **Debug > Start without debugging** or type **Ctrl + F5**. If the program compiles correctly, Visual Studio opens a Command Prompt window and executes the program. We changed the Command Prompt's color scheme and font size for readability:

```
C:\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\cpp20_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

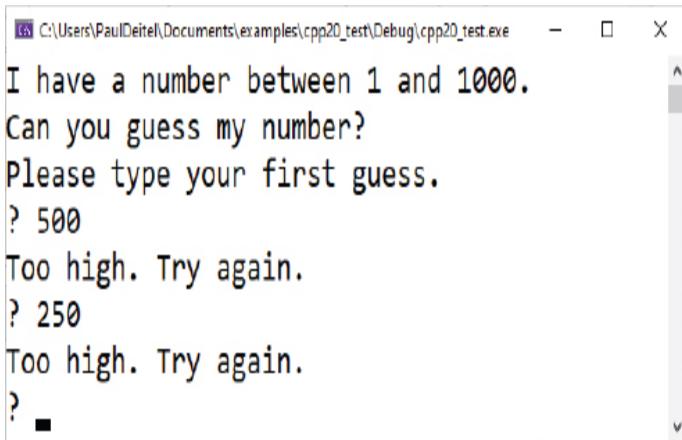
### Step 7: Entering Your First Guess

At the **?** prompt, type **500** and press *Enter*. The application displays "Too high. Try again." to indicate the value you entered is greater than the number the application chose as the correct guess:

```
C:\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\cpp20_test.exe
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

### Step 8: Entering Another Guess

At the next prompt, type **250** and press *Enter*. The application displays "Too high. Try again.", because the value you entered once again is greater than the correct guess:



A screenshot of a Windows command-line window titled "C:\Users\PaulDeitel\Documents\examples\cpp20\_test\Debug\cpp20\_test.exe". The window contains the following text:

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
? 500  
Too high. Try again.  
? 250  
Too high. Try again.  
?
```

### Step 9: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays "Excellent! You guessed the number!":

The screenshot shows a Windows command-line window titled "C:\Users\PaulDeitel\Documents\examples\cpp20\_test\Debug\cpp20\_test.exe". The application is running a number-guessing game. It starts by saying "I have a number between 1 and 1000." and asks "Can you guess my number?". The user types "500", and the application responds "Too high. Try again.". The user then types "250", and the application says "Too high. Try again.". This continues with the user's guesses being 125, 63, 31, 47, 39, 43, 41, and finally 42, each followed by a "Too high. Try again." message. After the user enters 42, the application says "Excellent! You guessed the number!" and then asks "Would you like to play again (y or n)?".

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
? 500  
Too high. Try again.  
? 250  
Too high. Try again.  
? 125  
Too high. Try again.  
? 63  
Too high. Try again.  
? 31  
Too low. Try again.  
? 47  
Too high. Try again.  
? 39  
Too low. Try again.  
? 43  
Too high. Try again.  
? 41  
Too low. Try again.  
? 42  
  
Excellent! You guessed the number!  
Would you like to play again (y or n)?
```

### Step 10: Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application. Each time you execute this application from the beginning (*Step 6*), it will choose the same numbers for you to guess.

### Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. However, for our examples, you may find it more convenient to remove the current program from the project then add a new one. To remove a file from your project (but not your system), select it in the **Solution Explorer** then press *Del* (or *Delete*). You can then repeat *Step 4* to add a different program to the project.

### 1.2.2 Compiling and Running a C++20 Application with Xcode on macOS

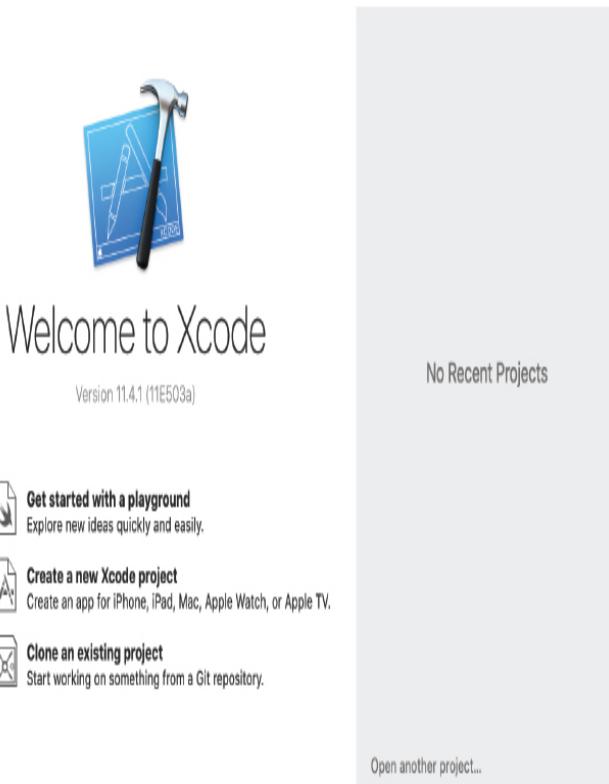
In this section, you'll run a C++ program on a macOS using the Clang compiler in Apple's Xcode IDE.

#### Step 1: Checking Your Setup

If you have not already done so, read the Before You Begin section of this book for instructions on installing the IDE and downloading the book's code examples.

#### Step 2: Launching Xcode

Open a Finder window, select **Applications** and double-click the Xcode icon (  ). If this is your first time running Xcode, the **Welcome to Xcode** window appears:



Close this window by clicking the **X** in the upper left corner—you can access it any time by selecting **Window > Welcome to Xcode**. We use the **>** character to indicate selecting a menu item from a menu. For example, the notation **File > Open...** indicates that you should select the **Open...** menu item from the **File** menu.

### Step 3: Creating a Project

A **project** is a group of related files, such as the C++ source-code files that compose an application. The Xcode projects we created for this book’s examples are **Command Line Tool** projects that you’ll execute directly in the IDE. To create a project:

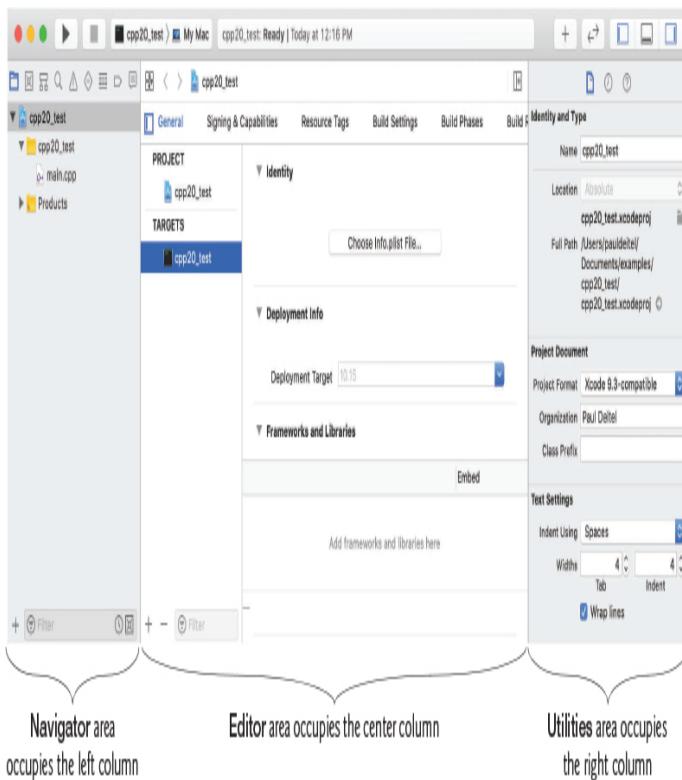
1. Select **File > New > Project....**
2. At the top of the **Choose a template for your new project** dialog, click **macOS**.
3. Under **Application**, click **Command Line Tool** and click **Next**.
4. For **Product Name**, enter a name for your project—we specified `cpp20_test`.

**5.** In the **Language** drop-down list, select **C++** then click **Next**.

**6.** Specify where you want to save your project. We selected the **examples** folder containing this book’s code examples.

**7. Click Create.**

Xcode creates your project and displays the **workspace window** initially showing three areas—the **Navigator area**, **Editor area** and **Utilities area**:



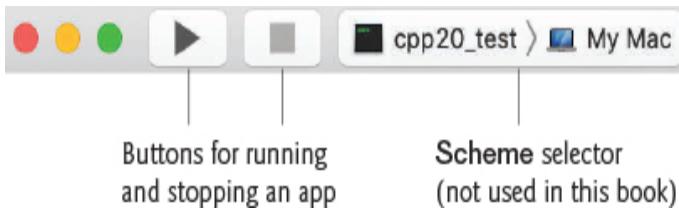
The left-side **Navigator** area has icons at its top for the *navigators* that can be displayed there. For this book, you’ll primarily work with

- **Project** (📁)—Shows all the files and folders in your project.
- **Issue** (⚠️)—Shows you warnings and errors generated by the compiler.

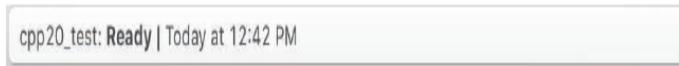
Clicking a navigator button displays the corresponding navigator panel.

The middle **Editor** area is for managing project settings and editing source code. This area is always displayed in your workspace window. Selecting a file in the **Project** navigator, the file's contents display in the **Editor** area. You will not use the right-side **Utilities** area in this book. There's also a **Debug area** in which you'll run and interact with the guess-the-number program. This will appear below the **Editor** area.

The workspace window's toolbar contains options for executing a program:



displaying the progress of tasks executing in Xcode:



and hiding or showing the left (Navigator), right (Utilities) and bottom (Debug) areas:



#### Step 4: Configuring the Project to Compile Using C++20

The Clang compiler in Xcode supports several versions of the C++ standard. For this book, we use C++20, which we must configure in our project's settings:

1. In the **Project** navigator, select your project's name (`cpp20_test`).
2. In the **Editors** area's left side, select your project's name under **TARGET**.
3. At the top of the **Editors** area, click **Build Settings**, and just below it, click **All**.
4. Scroll to the **Apple Clang - Language - C++** section.
5. Click the value to the right of **C++ Language Dialect** and select **Other....**

6. In the popup area, replace the current setting with `c++2a` and press *Enter*. In a future version of Xcode, Apple will provide a C++20 option for **C++ Language Dialect**.

### Step 5: Deleting the `main.cpp` File from the Project

By default, Xcode creates a `main.cpp` source-code file containing a simple program that displays "Hello, World!". You won't use `main.cpp` in this test-drive, so you should delete the file. In the **Project** navigator, right-click the `main.cpp` file and select **Delete**. In the dialog that appears, select **Move to Trash**. The file will not be removed from your system until you empty your trash.

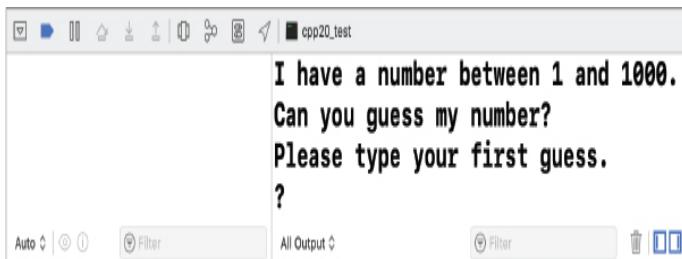
### Step 6: Adding the `GuessNumber.cpp` File into the Project

In a Finder window, open the `ch01` folder in the book's `examples` folder, then drag `GuessNumber.cpp` onto the **Guess Number** folder in the **Project** navigator. In the dialog that appears, ensure that **Copy items if needed** is checked, then click **Finish**.<sup>3</sup>

3. For the multiple source-code-file programs that you'll see later in the book, drag all the files for a given program to the project's folder. When you begin creating your own programs, you can right click the project's folder and select **New File...** to display a dialog for adding a new file.

### Step 7: Compiling and Running the Project

To compile and run the project so you can test-drive the application, simply click the run (▶) button on Xcode's toolbar. If the program compiles correctly, Xcode opens the **Debug** area and executes the program in the right half of the **Debug** area:



The application displays "Please type your first guess.", then displays a question mark (?) as a prompt on the next line.

### Step 8: Entering Your First Guess

Click in the **Debug** area, then type **500** and press *Return*

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
? 500  
Too low. Try again.  
?
```

The application displays "Too low. Try again.", meaning that the value you entered is less than the number the application chose as the correct guess.

### Step 9: Entering Another Guess

At the next prompt, enter **750**:

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
? 500  
Too low. Try again.  
? 750  
Too low. Try again.  
?
```

The application displays "Too low. Try again.", because the value you entered once again is less than the correct guess.

### Step 10: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays "Excellent! You guessed the number.":

```
? 875
Too high. Try again.
? 812
Too high. Try again.
? 781
Too low. Try again.
? 797
Too low. Try again.
? 805
Too low. Try again.
? 808

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

### Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application. Each time you execute this application from the beginning (*Step 7*), it will choose the same numbers for you to guess.

### Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. However, for our examples, you may find it more convenient to remove the current program from the project then add a new one. To remove a file from your project (but not your system), right-click the file in the **Project** navigator and select **Delete**. In the dialog that appears, select **Move to Trash**. You can then repeat *Step 6* to add a different program to the project.

## 1.2.3 Compiling and Running a C++20 Application with GNU C++ on Linux

For this test drive, we assume that you read the Before You Begin section and that you placed the downloaded examples in your user account's **Documents** folder.

### Step 1: Changing to the `ch01` Folder

From a Linux shell, use the `cd` command to change to the `ch01` subfolder of the book's `examples` folder:

```
~$ cd ~/Documents/examples/ch01  
~/Documents/examples/ch01$
```

In this section's figures, we use **bold** to highlight the text that you type. The prompt in our Ubuntu Linux shell uses a tilde (~) to represent the home directory. Each prompt ends with the dollar sign (\$). The prompt may differ on your Linux system.

## Step 2: Compiling the Application

Before running the application, you must first compile it:

```
~/Documents/examples/ch01$ g++ -std=c++2a GuessNumber.  
~/Documents/examples/ch01$
```

The `g++` command compiles the application:

- The `-std=c++2a` option indicates that we're using C++20—`c++2a` will become `c++20` in a future GNU C++ release.
- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program.

## Step 3: Running the Application

Type `./GuessNumber` at the prompt and press *Enter* to run the program:

```
~/Documents/examples/ch01$ ./GuessNumber  
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
?
```

The `.` tells Linux to run a file from the current directory and is required to indicate that `GuessNumber` is an executable file.

## Step 4: Entering Your First Guess

The application displays "Please type your first guess.", then displays a question mark (?) as a prompt on the next line. At the prompt, enter **500**—note that the outputs may vary based on the compiler you're using:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

The application displays "Too high. Try again.", meaning that the value you entered is greater than the number the application chose as the correct guess.

### Step 5: Entering Another Guess

At the next prompt, enter **250**:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

This time the application displays "Too low. Try again.", because the value you entered is less than the correct guess.

### Step 6: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays "Excellent! You guessed the number.":

```
Too low. Try again.
? 375
Too low. Try again.
? 437
```

```
Too high. Try again.  
? 406  
Too high. Try again.  
? 391  
Too high. Try again.  
? 383  
Too low. Try again.  
? 387  
Too high. Try again.  
? 385  
Too high. Try again.  
? 384  
Excellent! You guessed the number.  
Would you like to play again (y or n)?
```

## Step 7: Playing the Game Again or Exiting the Application

After you guess the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application and returns you to the shell. Each time you execute this application from the beginning (*Step 3*), it will choose the same numbers for you to guess.

### 1.2.4 Compiling and Running a C++20 Application with GNU C++ in the GCC Docker Container in Docker Running Natively Over Windows 10, macOS and/or Linux

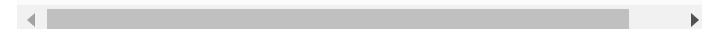
At the time of this writing, GNU C++ implements the most C++20 features. For this reason, you may want to use the latest GNU C++ compiler on your system. One of the most convenient cross-platform ways to do this is by using the GNU Compiler Collection (GCC) Docker container. This section assumes you've already installed Docker Desktop (Windows or macOS) or Docker Engine (Linux).

#### Executing the GNU Compiler Collection (GCC) Docker Container

Open a Command Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then perform the following steps to launch the GCC Docker Container:

1. Use the `cd` command to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the GCC docker container with the command

```
docker run --rm -it -v "%CD%":/usr/src gcc:lates
```



3. macOS/Linux users: Launch the GCC docker container with the command

```
docker run --rm -it -v "$(pwd)":/usr/src gcc:lat
```

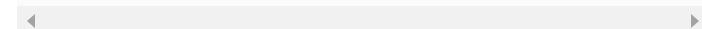


In the preceding commands:

- `--rm` cleans up the container's resources when you eventually shut it down.
- `-it` runs the container in interactive mode, so you can enter commands to change folders and to compile and run programs using the GNU C++ compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) allows the Docker container to access your local system files in the folder from which you executed the `docker run` command. In the Docker container, you'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the C++ code.
- `gcc:latest` is the container name. The `:latest` specifies that you want to use the most up-to-date version of the `gcc` container. Each time you execute the preceding `docker run` commands, Docker checks whether you have the latest `gcc` container version. If not, Docker downloads it before executing the container.

Once the container is running, you'll see a prompt like:

```
root@67773f59d9ea:/#
```



The container uses a Linux operating system. It's prompt displays the current folder location between the : and #.

### Changing to the ch01 Folder in the Docker Container

The docker run command specified above attaches your examples folder to the containers /usr/src folder. In the docker container, use the cd command to change to the ch01 sub-folder of /usr/src:

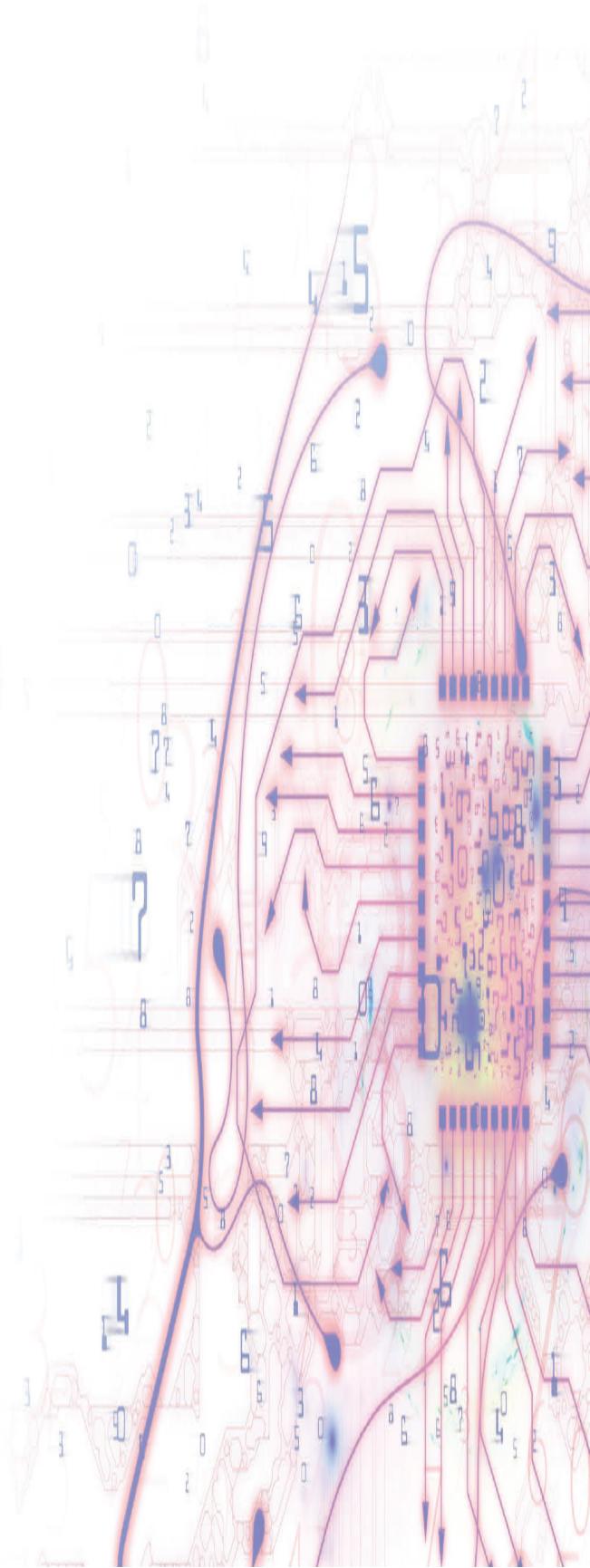
```
root@01b4d47cadc6:/# cd /usr/src/ch01
root@01b4d47cadc6:/usr/src/ch01#
```

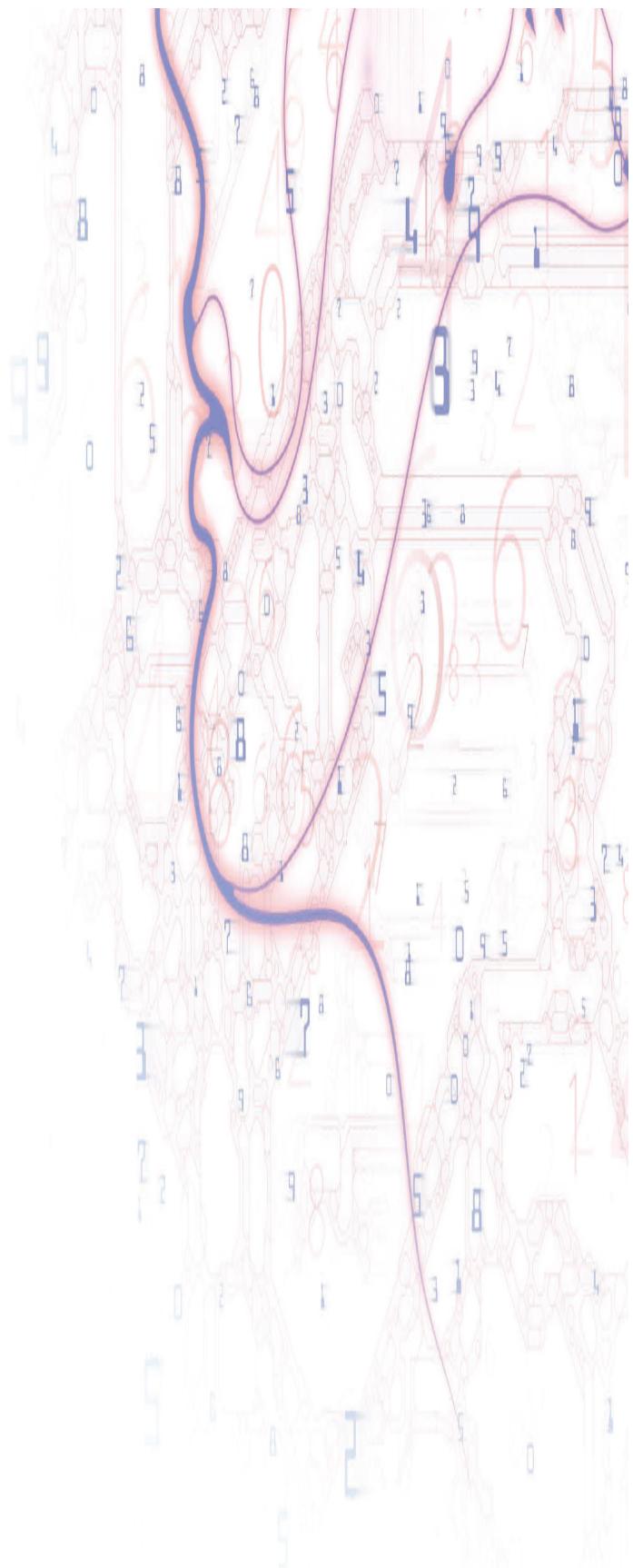
To compile, run and interact with the GuessNumber application in the Docker container, follow *Steps 2–7* of Section 1.2.3's GNU C++ Test-Drive.

### Terminating the Docker Container

You can terminate the Docker container by typing *Ctrl + d* at the container's prompt.

## 2. Intro to C++ Programming





## Objectives

In this chapter, you'll:

- Write simple C++ applications.
  - Use input and output statements.
  - Use fundamental data types.
  - Use arithmetic operators.
  - Understand the precedence of arithmetic operators.
  - Write decision-making statements.
  - Use relational and equality operators.
  - Begin appreciating the “objects natural” learning approach by creating and using objects of the C++ standard library’s `string` class before creating your own custom classes.
- 

## Outline

- [2.1 Introduction](#)
  - [2.2 First Program in C++: Displaying a Line of Text](#)
  - [2.3 Modifying Our First C++ Program](#)
  - [2.4 Another C++ Program: Adding Integers](#)
  - [2.5 Arithmetic](#)
  - [2.6 Decision Making: Equality and Relational Operators](#)
  - [2.7 Objects Natural: Creating and Using Objects of Standard Library Class `string`](#)
  - [2.8 Wrap-Up](#)
- 

### 2.1 INTRODUCTION

This chapter presents several code examples that demonstrate how your programs can display messages and obtain data from the user for processing. The first three display messages on the screen. The next obtains two numbers from a user at the keyboard, calculates their sum and displays the result—the accompanying discussion introduces C++’s arithmetic

operators. The fifth example demonstrates decision making by showing you how to compare two numbers, then display messages based on the comparison results.

### “Objects Natural” Learning Approach

In your programs, you’ll create and use many objects of carefully-developed-and-tested preexisting classes that enable you to perform significant tasks with minimal code. These classes typically come from:

- the C++ standard library,
- platform-specific libraries (such as those provided by Microsoft for creating Windows applications or by Apple for creating macOS applications) and
- free third-party libraries often created by the massive open-source communities that have developed around all major contemporary programming languages.

To help you appreciate this style of programming early in the book, you’ll create and use objects of preexisting C++ standard library classes before creating your own custom classes. We call this the “objects natural” approach. You’ll begin by creating and using `string` objects in this chapter’s final example. In later chapters, you’ll create your own custom classes. You’ll see that C++ enables you to “craft valuable classes” for your own use and that other programmers can reuse.

### Compiling and Running Programs

For instructions on compiling and running programs in Microsoft Visual Studio, Apple Xcode and GNU C++, see the test-drives in [Chapter 1](#) or our video instructions at

---

<http://deitel.com/c-plus-plus-20-for-programmers>



### “Rough-Cut” E-Book for O’Reilly Online Learning Subscribers

You are viewing an early-access “rough cut” of *C++20 for Programmers*. **We prepared this content carefully, but it has**

**not yet been reviewed or copy edited and is subject to change.** As we complete each chapter, we'll post it here. Please send any corrections, comments, questions and suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I'll respond promptly. Check here frequently for updates.

### **“Sneak Peek” Videos for O’Reilly Online Learning Subscribers**

As an O'Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

<https://learning.oreilly.com/videos/c-20-fundame>



Co-author Paul Deitel immediately records each video lesson as we complete each rough-cut e-book chapter. Lessons go live on O'Reilly Online Learning a few days later. Again, check here frequently for updates.

## **2.2 FIRST PROGRAM IN C++: DISPLAYING A LINE OF TEXT**

Consider a simple program that displays a line of text (Fig. 2.1). The line numbers are not part of the program.

---

```
1 // fig02_01.cpp
2 // Text-printing program.
3 #include <iostream> // enables program to output data to the screen
4
5 // function main begins program execution
6 int main() {
7     std::cout << "Welcome to C++!\n"; // display message
8
9     return 0; // indicate that program ended successfully
10 } // end function main
```

Welcome to C++!

**Fig. 2.1** Text-printing program.

## Comments

Lines 1 and 2

```
// fig02_01.cpp  
// Text-printing program.
```

each begin with **//**, indicating that the remainder of each line is a **comment**. In each of our programs, the first line comment contains the program's file name. The comment "Text - printing program." describes the purpose of the program. A comment beginning with // is called a **single-line comment** because it terminates at the end of the current line. You can create **multiline comments** by enclosing them in /\* and \*/ , as in

```
/* fig02_01.cpp  
Text-printing program. */
```

## #include Preprocessing Directive

Line 3

```
#include <iostream> // enables program to output
```

is a **preprocessing directive**—that is, a message to the C++ preprocessor, which the compiler invokes before compiling the program. This line notifies the preprocessor to include in the program the contents of the **input/output stream header <iostream>**. This header is a file containing information the compiler requires when compiling any program that outputs data to the screen or inputs data from the keyboard using C++'s stream input/output. The program in Fig. 2.1 outputs data to the screen. Chapter 5 discusses headers in more detail, and Chapter 15 explains the contents of <iostream> in more detail.

## Blank Lines and White Space

Line 4 is simply a blank line. You use blank lines, spaces and tabs to make programs easier to read. Together, these characters are known as **white space**—they’re normally ignored by the compiler.

## The `main` Function

Line 6

```
int main() {
```

is a part of every C++ program. The parentheses after **main** indicate that it’s a **function**. C++ programs typically consist of one or more functions and classes. Exactly one function in every program must be named **main**. Figure 2.1 contains only one function. C++ programs begin executing at function **main**. The keyword **int** to the left of **main** indicates that after **main** finishes executing, it “returns” an integer (whole number) value. **Keywords** are reserved by C++ for a specific use. We show the complete list of C++ keywords in Chapter 3. We’ll explain what it means for a function to “return a value” when we demonstrate how to create your own functions in Chapter 5. For now, simply include the keyword **int** to the left of **main** in each of your programs.

The **left brace**, **{**, (end of line 6) must *begin* each function’s **body**, which contains the instructions the function performs. A corresponding **right brace**, **}**, (line 10) must *end* each function’s body.

## An Output Statement

Line 7

```
std::cout << "Welcome to C++!\n"; // display mes
```

displays the characters contained between the double quotation marks. Together, the quotation marks and the characters between them are called a **string**, a **character string** or a **string literal**. We refer to characters between double quotation

marks simply as strings. White-space characters in strings are not ignored by the compiler.

The entire line 7—including `std::cout`, the `<< operator`, the string "Welcome to C++!\n" and the `semicolon (;`)—is called a **statement**. Most C++ statements end with a semicolon. Omitting the semicolon at the end of a C++ statement when one is needed is a syntax error. Preprocessing directives (such as `#include`) are not C++ statements and do not end with a semicolon.

Typically, output and input in C++ are accomplished with **streams** of data. When the preceding statement executes, it sends the stream of characters `Welcome to C++!\n` to the **standard output stream object (`std::cout`)**, which is normally “connected” to the screen.

## Indentation

Indent the body of each function one level within the braces that delimit the function’s body. This makes a program’s functional structure stand out, making the program easier to read. Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We prefer three spaces per level of indent.

## The `std` Namespace

The `std::` before `cout` is required when we use names that we’ve brought into the program by preprocessing directives like `#include <iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to namespace `std`. The names `cin` (the standard input stream) and `cerr` (the standard error stream)—introduced in [Chapter 1](#)—also belong to namespace `std`. We discuss namespaces in Chapter 23, Other Topics. For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and `cerr` in a program. This can be cumbersome—we’ll soon introduce `using` declarations and

the `using` directive, which will enable you to omit `std::` before each use of a name in the `std` namespace.

## The Stream Insertion Operator and Escape Sequences

In a `cout` statement, the `<<` operator is referred to as the **stream insertion operator**. When this program executes, the value to the operator's right (the right **operand**) is inserted in the output stream. Notice that the `<<` operator points toward where the data goes. A string's characters normally display exactly as typed between the double quotes. However, the characters `\n` are *not* displayed in Fig. 2.1's sample output. The backslash (`\`) is called an **escape character**. It indicates that a “special” character is to be output. When a backslash is encountered in a string, the next character is combined with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. It causes the **cursor** (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. Some common escape sequences are shown in the following table:

Escape sequence	Description
<code>\n</code>	Newline. Positions the screen cursor to the beginning of the next line.
<code>\t</code>	Horizontal tab. Moves the screen cursor to the next tab stop.
<code>\r</code>	Carriage return. Positions the screen cursor to the beginning of the current line; does not advance to the next line.
<code>\a</code>	Alert. Sound the system bell.
<code>\\\</code>	Backslash. Includes a backslash character in a string.
<code>\'</code>	Single quote. Includes a single-quote character in a string.
<code>\"</code>	Double quote. Includes a double-quote character in a string.

## The `return` Statement

Line 9

```
return 0; // indicate that program ended success
```



is one of several means we'll use to **exit a function**. In this **return statement** at the end of `main`, the value `0` indicates that the program terminated successfully. If program execution reaches the end of `main` without encountering a `return` statement, C++ assumes that the program terminated successfully. So, we omit the `return` statement at the end of `main` in subsequent programs that terminate successfully.

## 2.3 MODIFYING OUR FIRST C++ PROGRAM

The next two examples modify the program of Fig. 2.1. The first displays text on one line using multiple statements. The second displays text on several lines using one statement.

### Displaying a Single Line of Text with Multiple Statements

Figure 2.2 performs stream insertion in multiple statements (lines 7–8), yet produces the same output as Fig. 2.1. Each stream insertion resumes displaying where the previous one stopped. Line 7 displays `Welcome` followed by a space, and because this string did not end with `\n`, line 8 begins displaying on the same line immediately following the space.

---

```
1 // fig02_02.cpp
2 // Displaying a line of text with multiple statements.
3 #include <iostream> // enables program to output data to the screen
4
5 // function main begins program execution
6 int main() {
7     std::cout << "Welcome ";
8     std::cout << "to C++!\n";
9 } // end function main
```

Welcome to C++!

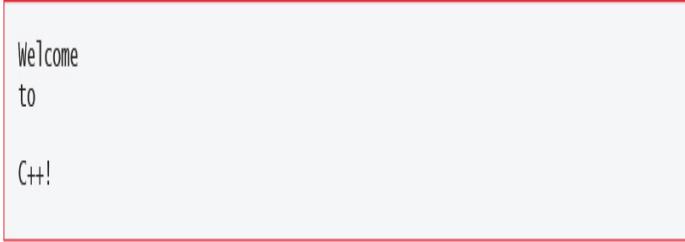
**Fig. 2.2** Displaying a line of text with multiple statements.

### Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using additional newline characters, as in line 7 of Fig. 2.3. Each time the `\n` (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, place two newline characters back to back, as in line 7.

---

```
1 // fig02_03.cpp
2 // Displaying multiple lines of text with a single statement.
3 #include <iostream> // enables program to output data to the screen
4
5 // function main begins program execution
6 int main() {
7     std::cout << "Welcome\ninto\n\nC++!\n";
8 } // end function main
```



```
Welcome
to
C++!
```

**Fig. 2.3** Displaying multiple lines of text with a single statement. (Part 1 of 2.)

## 2.4 ANOTHER C++ PROGRAM: ADDING INTEGERS

Our next program obtains two integers typed by a user at the keyboard, computes their sum and outputs the result using `std::cout`. Figure 2.4 shows the program and sample inputs and outputs. In the sample execution, the user's input is in **bold**.

```
1 // fig02_04.cpp
2 // Addition program that displays the sum of two integers.
3 #include <iostream> // enables program to perform input and output
4
5 // function main begins program execution
6 int main() {
7     // declaring and initializing variables
8     int number1{0}; // first integer to add (initialized to 0)
9     int number2{0}; // second integer to add (initialized to 0)
10    int sum{0}; // sum of number1 and number2 (initialized to 0)
11
12    std::cout << "Enter first integer: "; // prompt user for data
13    std::cin >> number1; // read first integer from user into number1
14
15    std::cout << "Enter second integer: "; // prompt user for data
16    std::cin >> number2; // read second integer from user into number2
17
18    sum = number1 + number2; // add the numbers; store result in sum
19
20    std::cout << "Sum is " << sum << std::endl; // display sum; end line
21 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

**Fig. 2.4** Addition program that displays the sum of two integers.

## Variable Declarations and List Initialization

Lines 8–10

```
int number1{0}; // first integer to add (initial
int number2{0}; // second integer to add (initia
int sum{0}; // sum of number1 and number2 (initi
```

are **declarations**—number1, number2 and sum are the names of **variables**. These declarations specify that the variables number1, number2 and sum are data of type **int**,

meaning that these variables will hold **integer** (whole number) values, such as 7, -11, 0 and 31914. All variables must be declared with a name and a data type.

**11** Lines 8–10 initialize each variable to 0 by placing a value in braces ({ and }) immediately following the variable's name—this is known as **list initialization**, which was introduced in C++11. Although it's not always necessary to initialize every variable explicitly, doing so will help you avoid many kinds of problems.

Prior to C++11, lines 8–10 would have been written as:

```
int number1 = 0; // first integer to add (initialization)
int number2 = 0; // second integer to add (initialization)
int sum = 0; // sum of number1 and number2 (initialization)
```

If you work with legacy C++ programs, you're likely to encounter initialization statements using this older C++ coding style. In subsequent chapters, we'll discuss various list initialization benefits.

### Declaring Multiple Variables at Once

Several variables of the same type may be declared in one declaration—for example, we could have declared and initialized all three variables in one declaration by using a comma-separated list as follows:

```
int number1{0}, number2{0}, sum{0};
```

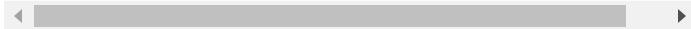
But, this makes the program less readable and makes it awkward to provide comments that describe each variable's purpose.

### Fundamental Types

We'll soon discuss the type **double** for specifying real numbers and the type **char** for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and -11.19. A **char** variable may hold only a single lowercase letter, uppercase letter, digit or special character (e.g., \$ or \*).

Types such as `int`, `double` and `char` are called **fundamental types**. Fundamental-type names consist of one or more keywords and must appear in all lowercase letters. For a complete list of C++ fundamental types and their typical ranges, see

<https://en.cppreference.com/w/cpp/language/types>



## Identifiers

A variable name (such as `number1`) is any valid **identifier** that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (`_`) that does not begin with a digit. C++ is **case sensitive**—uppercase and lowercase letters are different, so `a1` and `A1` are different identifiers.

C++ allows identifiers of any length, but some C++ implementations may restrict identifier lengths. Do not use identifiers that begin with underscores and double underscores, because C++ compilers use names like that for their own purposes internally.

## Placement of Variable Declarations

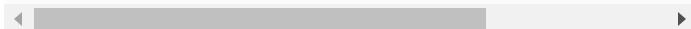
Variable declarations can be placed almost anywhere in a program, but they must appear before the variables are used. For example, the declaration in line 8

```
int number1{0}; // first integer to add (initial
```



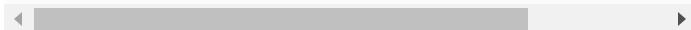
could have been placed immediately before line 13

```
std::cin >> number1; // read first integer from
```



the declaration in line 9

```
int number2{0}; // second integer to add (initia
```



could have been placed immediately before line 16

```
std::cin >> number2; // read second integer from
```



and the declaration in line 10

```
int sum{0}; // sum of number1 and number2 (initi
```



could have been placed immediately before line 18

```
sum = number1 + number2; // add the numbers; sto
```



In fact, lines 10 and 18 could have been combined into the following declaration and placed just before line 20:

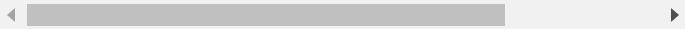
```
int sum{number1 + number2}; // initialize sum wi
```



## Obtaining the First Value from the User

Line 12

```
std::cout << "Enter first integer: "; // prompt
```



displays `Enter first integer:` followed by a space.

This message is called a **prompt** because it directs the user to take a specific action. Line 13

```
std::cin >> number1; // read first integer from
```



uses the **standard input stream object `cin`** (of namespace `std`) and the **stream extraction operator, `>>`**, to obtain a value from the keyboard.

When the preceding statement executes, the computer waits for the user to enter a value for variable `number1`. The user responds by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the computer. The computer converts the character representation of the number to an integer value and

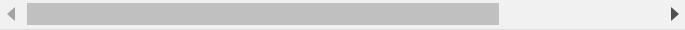
assigns this value to the variable `number1`. Pressing *Enter* also causes the cursor to move to the beginning of the next line on the screen.

When your program is expecting the user to enter an integer, the user could enter alphabetic characters, special symbols (like # or @) or a number with a decimal point (like 73.5), among others. In these early programs, we assume that the user enters valid data. We'll present various techniques for dealing with data-entry problems later.

### Obtaining the Second Value from the User

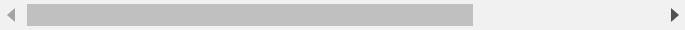
Line 15

```
std::cout << "Enter second integer: "; // prompt
```



displays `Enter second integer:` on the screen, prompting the user to take action. Line 16

```
std::cin >> number2; // read second integer from
```



obtains a value for variable `number2` from the user.

### Calculating the Sum of the Values Input by the User

The assignment statement in line 18

```
sum = number1 + number2; // add the numbers; sto
```



adds the values of variables `number1` and `number2` and assigns the result to variable `sum` using the **assignment operator =**. Most calculations are performed in assignment statements. The = operator and the + operator are **binary operators** because each has two operands. For the + operator, the two operands are `number1` and `number2`. For the preceding = operator, the two operands are `sum` and the value of the expression `number1 + number2`. Placing spaces on either side of a binary operator makes the operator stand out and makes the program more readable.

## Displaying the Result

Line 20

```
std::cout << "Sum is " << sum << std::endl; // d
```

displays the character string "Sum is" followed by the numerical value of variable `sum` followed by `std::endl`—a **stream manipulator**. The name `endl` is an abbreviation for “end line” and belongs to namespace `std`. The `std::endl` stream manipulator outputs a new-line, then “flushes the output buffer.” This simply means that, on some systems where outputs accumulate in the machine until there are enough to “make it worthwhile” to display them on the screen, `std::endl` forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

The preceding statement outputs multiple values of different types. The stream insertion operator “knows” how to output each type of data. Using multiple stream insertion operators (`<<`) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**.

Calculations can also be performed in output statements. We could have combined the statements in lines 18 and 20 into the statement

```
std::cout << "Sum is " << number1 + number2 << s
```

thus eliminating the need for the variable `sum`.

The signature feature of C++ is that you can create your own data types called classes (we discuss this in [Chapter 10](#) and explore it in depth in [Chapter 11](#)). You can then “teach” C++ how to input and output values of these new data types using the `>>` and `<<` operators, respectively. This is called **operator overloading**, which we explore in [Chapter 14](#).

## 2.5 ARITHMETIC

The following table summarizes the **arithmetic operators**:

Operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$b m$ or $b \cdot m$	<code>b * m</code>
Division	/	$x / y$ or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

Note the use of various special symbols not used in algebra.

The **asterisk (\*)** indicates multiplication and the **percent sign (%)**

(%) is the remainder operator, which we'll discuss shortly.

These arithmetic operators are all binary operators.

### Integer Division

**Integer division** in which the numerator and the denominator are integers yields an integer quotient. For example, the expression `7/4` evaluates to 1, and the expression `17 / 5` evaluates to 3. Any fractional part in the result of integer division is truncated—no rounding occurs.

### Remainder Operator

The **remainder operator**, %, yields the remainder after integer division and can be used only with integer operands—`x % y` yields the remainder after dividing `x` by `y`. Thus, `7%4` yields 3 and `17 % 5` yields 2.

### Parentheses for Grouping Subexpressions

Parentheses are used in C++ expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c` we write `a * (b + c)`.

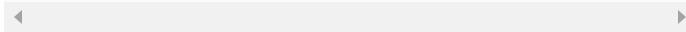
### Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence**, which are generally the same as those in algebra:

- 1.** Expressions in parentheses evaluate first. Parentheses are said to be at the “highest level of precedence.” In cases of **nested** or **embedded parentheses**, such as

---

```
(a * (b + c))
```



expressions in the innermost pair of parentheses evaluate first.

- 2.** Multiplication, division and remainder operations evaluate next. If an expression contains several multiplication, division and remainder operations, they’re applied from left-to-right. These three operators are said to be on the same level of precedence.
- 3.** Addition and subtraction operations evaluate last. If an expression contains several addition and subtraction operations, they’re applied from left-to-right. Addition and subtraction also have the same level of precedence.

Appendix A contains the complete operator precedence chart.

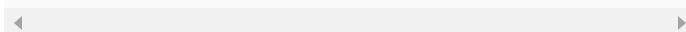
**Caution:** If you have an expression such as  $(a + b) * (c - d)$  in which two sets of parentheses are not nested, but appear “on the same level,” the C++ Standard does not specify the order in which these parenthesized subexpressions will evaluate.

## Operator Grouping

When we say that C++ applies certain operators from left-to-right, we are referring to the operators’ **grouping**. For example, in the expression

---

```
a + b + c
```



the addition operators (+) group from left-to-right as if we parenthesized the expression as  $(a+b)+ c$ . Most C++ operators of the same precedence group left-to-right. We’ll see that some operators group right-to-left.

## 2.6 DECISION MAKING: EQUALITY AND RELATIONAL OPERATORS

We now introduce C++'s **if statement**, which allows a program to take alternative action based on whether a **condition** is true or false. Conditions in **if** statements can be formed by using the **relational operators** and **equality operators** in the following table:

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
<i>Relational operators</i>			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	≥=	x ≥ y	x is greater than or equal to y
≤	≤=	x ≤ y	x is less than or equal to y
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

The relational operators all have the same level of precedence and group left-to-right. The equality operators both have the same level of precedence, which is lower than that of the relational operators, and group left-to-right.

Reversing the order of the pair of symbols in the operators !=, ≥= and ≤= (by writing them as =!, => and =<, respectively) is normally a syntax error. In some cases, writing != as =! will not be a syntax error, but almost certainly will be a logic error that has an effect at execution time. You'll understand why when we cover logical operators in [Chapter 4](#).

### Confusing == and =

Confusing the equality operator == with the assignment operator = results in logic errors. We like to read the equality operator as “is equal to” or “double equals,” and the assignment operator as “gets” or “gets the value of” or “is

assigned the value of.” As you’ll see in Section 4.12, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause subtle logic errors.

### Using the `if` Statement

Figure 2.5 uses six `if` statements to compare two integers input by the user. If a given `if` statement’s condition is true, the output statement in the body of that `if` statement executes. If the condition is false, the output statement in the body does not execute.

---

```
1 // fig02_05.cpp
2 // Comparing integers using if statements, relational operators
3 // and equality operators.
4 #include <iostream> // enables program to perform input and output
5
6 using std::cout; // program uses cout
7 using std::cin; // program uses cin
8 using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main() {
12     int number1{0}; // first integer to compare (initialized to 0)
13     int number2{0}; // second integer to compare (initialized to 0)
14
15     cout << "Enter two integers to compare: "; // prompt user for data
16     cin >> number1 >> number2; // read two integers from user
17
18     if (number1 == number2) {
19         cout << number1 << " == " << number2 << endl;
20     }
21
22     if (number1 != number2) {
23         cout << number1 << " != " << number2 << endl;
24     }
25
26     if (number1 < number2) {
27         cout << number1 << " < " << number2 << endl;
28     }
29
30     if (number1 > number2) {
31         cout << number1 << " > " << number2 << endl;
32     }
33
34     if (number1 <= number2) {
35         cout << number1 << " <= " << number2 << endl;
36     }
37
38     if (number1 >= number2) {
39         cout << number1 << " >= " << number2 << endl;
40     }
41 } // end function main
```

```
Enter two integers to compare: 3 7
```

```
3 != 7
```

```
3 < 7
```

```
3 <= 7
```

```
Enter two integers to compare: 22 12
```

```
22 != 12
```

```
22 > 12
```

```
22 >= 12
```

```
Enter two integers to compare: 7 7
```

```
7 == 7
```

```
7 <= 7
```

```
7 >= 7
```

**Fig. 2.5** Comparing integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

## using Declarations

Lines 6–8

```
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are **using declarations** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. We can now write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, respectively, in the remainder of the program.

## using Directive

In place of lines 6–8, many programmers prefer the **using directive**

```
using namespace std;
```

which, when you include a C++ standard library header (such as `<iostream>`), enables your program to use any name from that header's `std` namespace. From this point forward in the book, we'll use this directive in our programs.<sup>1</sup>

<sup>1</sup>. In Chapter 23, Other Topics, we'll discuss some issues with using directives in large-scale systems.

## Variable Declarations and Reading the Inputs from the User

Lines 12–13

```
int number1{0}; // first integer to compare (ini  
int number2{0}; // second integer to compare (in
```

declare the variables used in the program and initialize them to 0.

Line 16

```
cin >> number1 >> number2; // read two integers
```

uses cascaded stream extraction operations to input two integers. Recall that we're allowed to write `cin` (instead of `std::cin`) because of line 7. First, a value is read into `number1`, then a value is read into `number2`.

## Comparing Numbers

The `if` statement in lines 18–20

```
if (number1 == number2) {  
    cout << number1 << " == " << number2 << endl;  
}
```

determines whether the values of variables `number1` and `number2` are equal. If so, the `cout` statement displays a line of text indicating that the numbers are equal. For each condition that is `true` in the remaining `if` statements starting

in lines 22, 26, 30, 34 and 38, the corresponding cout statement displays an appropriate line of text.

### Braces and Blocks

Each if statement in Fig. 2.5 contains a single body statement that's indented to enhance readability. Also, notice that we've enclosed each body statement in a pair of braces, { }, creating what's called a **compound statement** or a **block**.

You don't need to use braces, { }, around single-statement bodies, but you must include the braces around multiple-statement bodies. Forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors, as a rule, always enclose an if statement's body statement(s) in braces.

### Common Logic Error: Placing a Semicolon After a Condition

Placing a semicolon immediately after the right parenthesis of the condition in an if statement is often a logic error (although not a syntax error). The semicolon causes the body of the if statement to be empty, so the if statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the if statement now becomes a statement in sequence with the if statement and always executes, often causing the program to produce incorrect results.

### Splitting Lengthy Statements

A lengthy statement may be spread over several lines. If a statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

### Operator Precedence and Grouping

With the exception of the assignment operator =, all the operators presented in this chapter group from left-to-right. Assignments (=) group from right-to-left. So, an expression such as  $x = y = \theta$  evaluates as if it had been written  $x =$

`(y = 0)`, which first assigns 0 to `y`, then assigns the result of that assignment (that is, 0) to `x`.

Refer to the complete operator precedence chart in [Appendix A](#) when writing expressions containing many operators.

Confirm that the operators in the expression are performed in the order you expect. If you’re uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you’d do in an algebraic expression.

## 2.7 OBJECTS NATURAL: CREATING AND USING OBJECTS OF STANDARD LIBRARY CLASS `STRING`

Throughout this book, we emphasize using preexisting valuable classes from the C++ standard library and various open-source libraries from the C++ open-source community. You’ll focus on knowing what libraries are out there, choosing the ones you’ll need for your applications, creating objects from existing library classes and making those objects exercise their capabilities. By Objects Natural, we mean that you’ll be able to program with powerful objects before you learn to create custom classes

You’ve already worked with C++ objects—specifically the `cout` and `cin` objects, which encapsulate the mechanisms for output and input, respectively. These objects were created for you behind the scenes using classes from the header `<iostream>`. In this section, you’ll create and interact with objects of the C++ standard library’s `string`<sup>2</sup> class.

2. You’ll learn additional `string` capabilities in subsequent chapters. [Chapter 8](#) discusses class `string` in detail, test-driving many more of its member functions.

### Test-Driving Class `string`

Classes cannot execute by themselves. A `Person` object can drive a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car’s internal mechanisms work. Similarly, the `main` function can “drive” a `string` object by calling its member functions

—without knowing how the class is implemented. In this sense, `main` in the following program is referred to as a **driver program**. Figure 2.6’s `main` function test-drives several `string` member functions.

---

```
1 // fig02_06.cpp
2 // Standard Library string class test program.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main() {
8     string s1{"happy"};
9     string s2{"birthday"};
10    string s3; // creates an empty string
11
12    // display the strings and show their lengths (Length is C++20)
13    cout << "s1: \"" << s1 << "\"; length: " << s1.length()
14    << "\ns2: \"" << s2 << "\"; length: " << s2.length()
15    << "\ns3: \"" << s3 << "\"; length: " << s3.length();
16
17    // compare strings with == and !=
18    cout << "\n\nThe results of comparing s2 and s1:" << boolalpha
19    << "\ns2 == s1: " << (s2 == s1)
20    << "\ns2 != s1: " << (s2 != s1);
21
22    // test string member function empty
23    cout << "\n\nTesting s3.empty():\n";
24}
```

```

25  if (s3.empty()) {
26      cout << "s3 is empty; assigning to s3;\n";
27      s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2
28      cout << "s3: \"" << s3 << "\"";
29  }
30
31 // testing new C++20 string member functions
32 cout << "\n\ns1 starts with \"ha\": " << s1.starts_with("ha") << endl;
33 cout << "s2 starts with \"ha\": " << s2.starts_with("ha") << endl;
34 cout << "s1 ends with \"ay\": " << s1.ends_with("ay") << endl;
35 cout << "s2 ends with \"ay\": " << s2.ends_with("ay") << endl;
36 }
```

s1: "happy"; length: 5  
s2: " birthday"; length: 9  
s3: ""; length: 0

The results of comparing s2 and s1:

s2 == s1: false  
s2 != s1: true

Testing s3.empty():

s3 is empty; assigning to s3;  
s3: "happy birthday"

s1 starts with "ha": true  
s2 starts with "ha": false  
s1 ends with "ay": false  
s2 ends with "ay": true

**Fig. 2.6** Standard Library `string` class test program.

(Part 1 of 2.)

## Instantiating Objects

Typically, you cannot call a member function of a class until you create an object of that class<sup>3</sup>—also called instantiating an object. Lines 8–10 create three `string` objects:

- `s1` is initialized with the string literal "happy",
- `s2` is initialized with the string literal " birthday"  
and

- `s3` is initialized by default to the **empty string** (that is, `""`).

3. You'll see in Section 11.15 that you can call a class's **static** member functions without creating an object of that class.

When we declare `int` variables, as we did earlier, the compiler knows what `int` is—it's a fundamental type that's built into C++. In lines 8–10, however, the compiler does not know in advance what type `string` is—it's a class type from the C++ standard library.

When packaged properly, classes can be reused by other programmers. This is one of the most significant benefits of working with object-oriented programming languages like C++ that have rich libraries of powerful prebuilt classes. For example, you can reuse the C++ standard library's classes in any program by including the appropriate headers—in this case, the **<string> header** (line 4). The name `string`, like the name `cout`, belongs to namespace `std`.

#### C++20 `string` Member Function `length`

**20** Lines 13–15 output each `string` and its length. The `string` class's **length member function** (new in C++20) returns the number of characters stored in a particular `string` object. In line 13, the expression

---

```
s1.length()
```



returns `s1`'s length by calling the object's `length` member function. To call this member function for a specific object, you specify the object's name (`s1`), followed by the **dot operator** (`.`), then the member function name (`length`) and a set of parentheses. *Empty* paren-theses indicate that `length` does not require any additional information to perform its task. Soon, you'll see that some member functions require additional information called arguments to perform their tasks.

From `main`'s view, when the `length` member function is called:

1. The program transfers execution from the call (line 13 in `main`) to member function `length`. Because `length` was called via the `s1` object, `length` “knows” which object’s data to manipulate.
2. Next, member function `length` performs its task—that is, it returns `s1`’s length to line 13 where the function was called. The `main` function does not know the details of how `length` performs its task, just as the driver of a car doesn’t know the details of how engines, transmissions, steering mechanisms and brakes are implemented.
3. The `cout` object displays the number of characters returned by member function `length`, then the program continues executing, displaying the `strings` `s2` and `s3` and their lengths.

## Comparing `string` Objects with the Equality Operators

Like numbers, `strings` can be compared with one another. Lines 18–20 show the results of comparing `s2` to `s1` using the equality operators—`string` comparisons are case sensitive.<sup>4</sup>

4. In Chapter 8, you’ll see that strings perform lexicographical comparisons using the numerical values of the characters in each string.

Normally, when you output a condition’s value, C++ displays `0` for false or `1` for true. The stream manipulator `boolalpha` (line 18) from the `<iostream>` header tells the output stream to display condition values as the words `false` or `true`.

### `string` Member Function `empty`

Line 25 calls `string` member function `empty`, which returns `true` if the `string` is empty; otherwise, it returns `false`. The object `s3` was initialized by default to the empty string, so it is indeed empty, and the body of the `if` statement will execute.

### `string` Concatenation and Assignment

Line 27 assigns a new value to `s3` produced by “adding” the strings `s1` and `s2` using the `+` operator—this is known as **string concatenation**. After the assignment, `s3` contains the characters of `s1` followed by the characters of `s2`. Line 28 outputs `s3` to demonstrate that the assignment worked correctly.

#### C++20 string Member Functions `starts_with` and `ends_with`

**20** Lines 32–35 demonstrate new `string` member functions `starts_with` and `ends_with`, which return `true` if the `string` starts with or ends with a specified substring, respectively; otherwise, they return `false`. Lines 32 and 33 show that `s1` starts with "ha", but `s2` does not. Lines 34 and 35 show that `s1` does not end with "ay" but `s2` does.

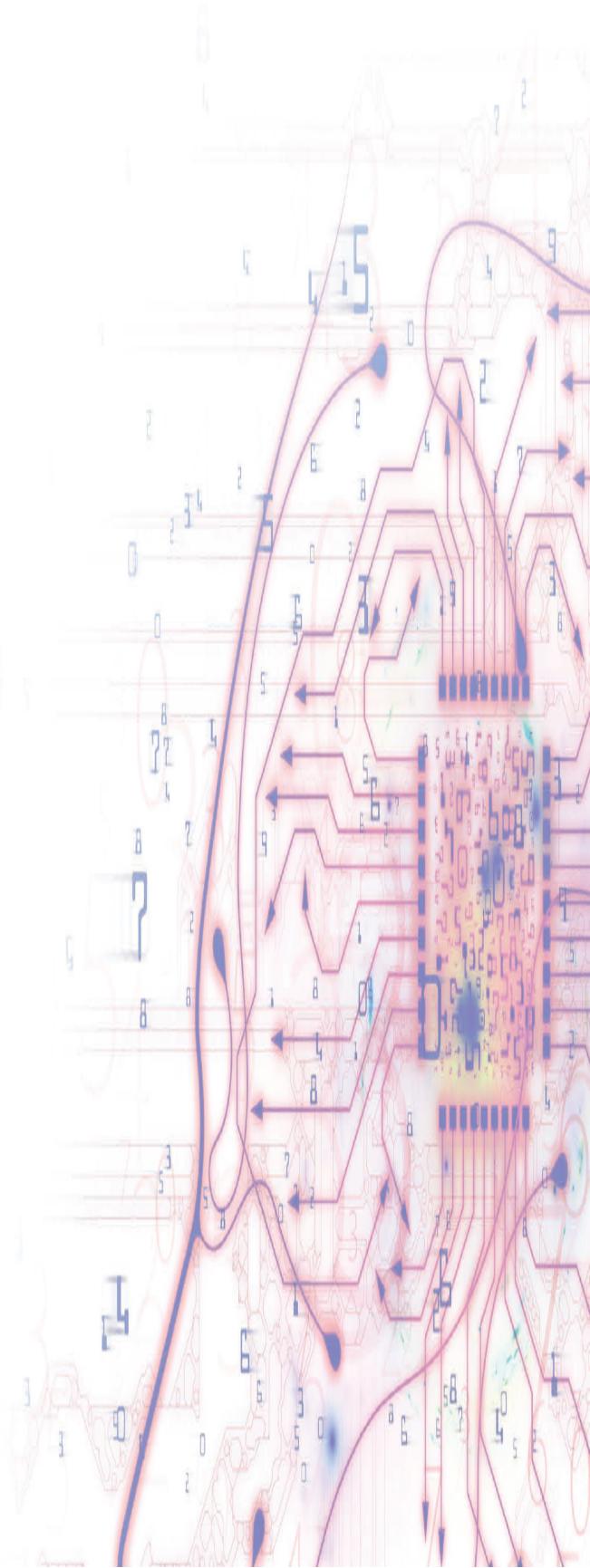
## 2.8 WRAP-UP

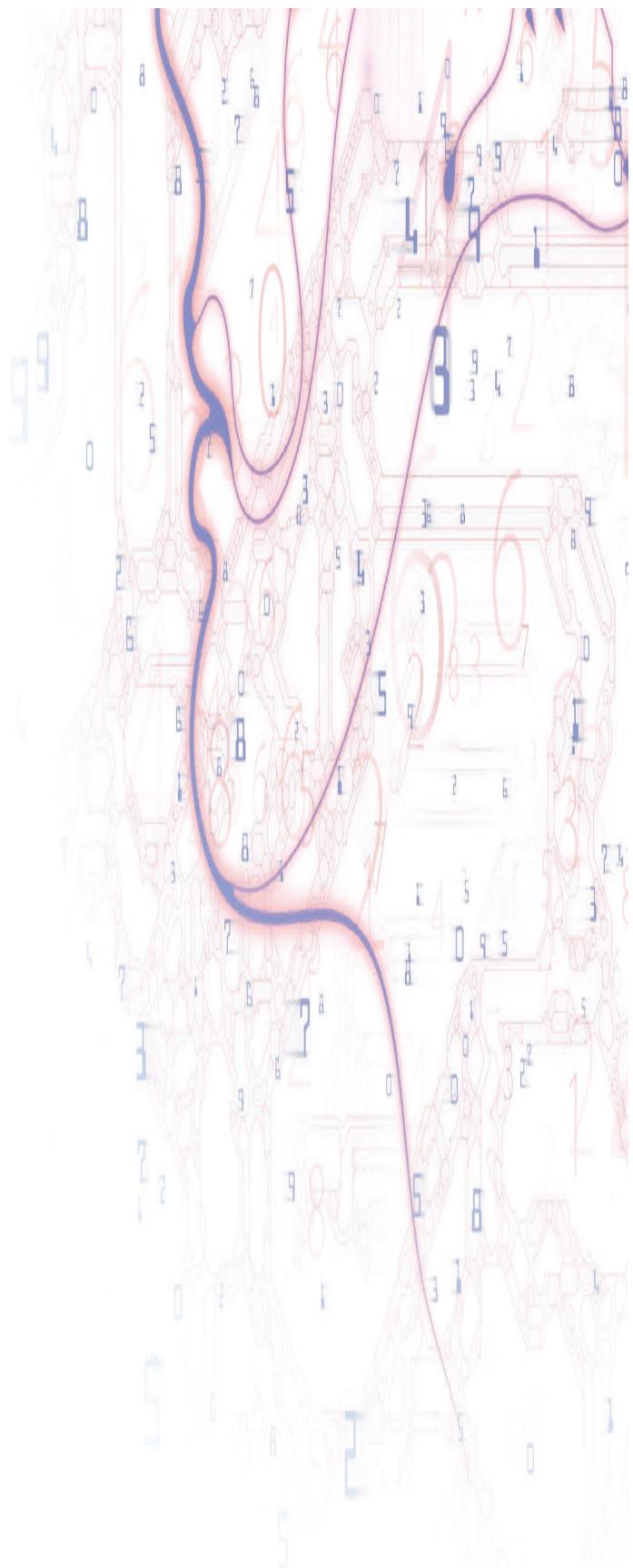
We presented many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. In particular, you learned to use the output stream object `cout` and the input stream object `cin` to build simple interactive programs. We declared and initialized variables and used arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the grouping of the operators. You saw how C++’s `if` statement allows a program to make decisions. We introduced the equality and relational operators, which we used to form conditions in `if` statements.

Finally, we introduced the notion of “objects natural” learning by creating objects of the C++ standard library class `string` and interacting with them using equality operators and `string` member functions. In subsequent chapters, you’ll create and use many objects of existing classes to accomplish significant tasks with minimal amounts of code. Then, in Chapters 10–14, you’ll create your own custom classes. You’ll

see that C++ enables you to “craft valuable classes.” In the next chapter, we begin our introduction to control statements, which specify the order in which a program’s actions are performed.

### **3. Control Statements: Part 1**





## Objectives

In this chapter, you'll:

- Use the `if` and `if...else` selection statements to choose between alternative actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use nested control statements.
- Use the compound assignment operators and the increment and decrement operators.
- Learn why fundamental data types are not portable.
- Continue learning with our objects natural approach with a case study on creating and manipulating integers as large as you want them to be.
- Use C++20's new text formatting capabilities, which are more concise and more powerful than those in earlier C++ versions.

---

## Outline

### [3.1 Introduction](#)

### [3.2 Control Structures](#)

#### [3.2.1 Sequence Structure](#)

#### [3.2.2 Selection Statements](#)

#### [3.2.3 Iteration Statements](#)

#### [3.2.4 Summary of Control Statements](#)

### [3.3 if Single-Selection Statement](#)

### [3.4 if...else Double-Selection Statement](#)

#### [3.4.1 Nested if...else Statements](#)

#### [3.4.2 Blocks](#)

#### [3.4.3 Conditional Operator \(?:\)](#)

### [3.5 while Iteration Statement](#)

### **3.6** Counter-Controlled Iteration

[3.6.1 Implementing Counter-Controlled Iteration](#)

[3.6.2 Integer Division and Truncation](#)

### **3.7** Sentinel-Controlled Iteration

[3.7.1 Implementing Sentinel-Controlled Iteration](#)

[3.7.2 Converting Between Fundamental Types](#)

[Explicitly and Implicitly](#)

[3.7.3 Formatting Floating-Point Numbers](#)

### **3.8** Nested Control Statements

[3.8.1 Problem Statement](#)

[3.8.2 Implementing the Program](#)

[3.8.3 Preventing Narrowing Conversions with C++11 List Initialization](#)

### **3.9** Compound Assignment Operators

### **3.10** Increment and Decrement Operators

### **3.11** Fundamental Types Are Not Portable

### **3.12** Objects Natural Case Study: Arbitrary Sized Integers

### **3.13** C++20 Feature Mock-Up—Text Formatting with Function `format`

### **3.14** Wrap-Up

---

## **3.1 INTRODUCTION**

In this chapter and the next, we present the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects. We discuss C++’s `if` statement in additional detail and introduce the `if...else`

and `while` statements. We also introduce the compound assignment operators and the increment and decrement operators.

**20** We discuss why C++’s fundamental types are not portable. We continue our object natural approach with a case study on arbitrary sized integers that support values beyond the ranges of integers supported by computer hardware.

We begin introducing C++20’s new text-formatting capabilities, which are based on those in Python, Microsoft’s .NET languages (like C# and Visual Basic) and Rust.<sup>1</sup> The C++20 capabilities are more concise and more powerful than those in earlier C++ versions. In Chapter 14, you’ll see that these new capabilities are extensible, so you can use them to format objects of custom class types.

1. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.htm>

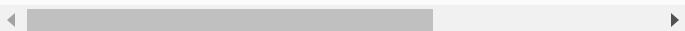
## “Rough-Cut” E-Book for O’Reilly Online Learning Subscribers

You are viewing an early-access “rough cut” of *C++20 for Programmers*. We prepared this content carefully, but it has not yet been reviewed or copy edited and is subject to change. As we complete each chapter, we’ll post it here. Please send any corrections, comments, questions and suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I’ll respond promptly. Check here frequently for updates.

## “Sneak Peek” Videos for O’Reilly Online Learning Subscribers

As an O’Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

<https://learning.oreilly.com/videos/c-20-fundame>



Co-author Paul Deitel immediately records each video lesson as we complete each rough-cut e-book chapter. Lessons go live on O’Reilly Online Learning a few days later. Again, check here frequently for updates.

## 3.2 CONTROL STRUCTURES

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of many problems experienced by software development groups. The blame was pointed at the **goto statement** (used in most programming

languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program.

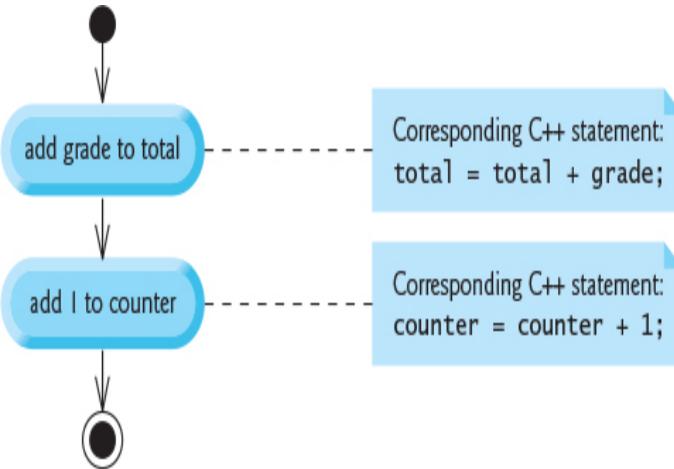
The research of Bohm and Jacopini<sup>2</sup> had demonstrated that programs could be written without any `goto` statements. The challenge for programmers of the era was to shift their styles to “`goto`-less programming.” The term **structured programming** became almost synonymous with “`goto` elimination.” The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug-free in the first place.

2. C. Bohm and G. Jacopini, “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules,” *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **iteration structure**. We’ll discuss how C++ implements each of these.

### 3.2.1 Sequence Structure

The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they appear in the program—that is, in sequence. The following UML<sup>3</sup> **activity diagram** illustrates a typical sequence structure in which two calculations are performed in order:



3. We use the UML in this chapter and [Chapter 4](#) to show the flow of control in control statements, then use UML again in [Chapters 10–13](#) when we present custom class development.

C++ lets you have as many actions as you want in a sequence structure. As you'll soon see, anywhere you may place a single action, you may place several actions in sequence.

An activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in the preceding diagram. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, which represent the flow of the activity—that is, the order in which the actions should occur.

The preceding sequence-structure activity diagram contains two **action states**, each containing an **action expression**—for example, “add grade to total” or “add 1 to counter”—that specifies a particular action to perform. The arrows in the activity diagram represent **transitions**, which indicate the order in which the actions represented by the action states occur.

The **solid circle** at the top of the activity diagram represents the **initial state**—the beginning of the workflow before the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram

represents the **final state**—that is, the end of the workflow after the program performs its actions.

The sequence structure activity diagram also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in C++)—explanatory remarks that describe the purpose of symbols in the diagram. A **dotted line** connects each note with the element it describes. We used the UML notes here to illustrate how the diagram relates to the C++ code for each action state. Activity diagrams usually do not show the C++ code.

### 3.2.2 Selection Statements

C++ has three types of **selection statements**. The **if** statement performs (selects) an action (or group of actions) if a condition is true, or skips it if the condition is false. The **if...else** statement performs an action (or group of actions) if a condition is true and performs a different action (or group of actions) if the condition is false. The **switch** statement (Chapter 4) performs one of many different actions (or group of actions), depending on the value of an expression.

The **if** statement is called a **single-selection statement** because it selects or ignores a single action (or group of actions). The **if...else** statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The **switch** statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

### 3.2.3 Iteration Statements

C++ provides four **iteration statements**—also called **repetition statements** or **looping statements**—for performing statements repeatedly while a **loop-continuation condition** remains true. The iteration statements are the **while**, **do...while**, **for** and range-based **for**. The **while** and **for** statements perform the action (or group of actions) in their bodies zero or more times. If the loop-continuation condition is initially false, the action (or group of actions) does not

execute. The `do...while` statement performs the action (or group of actions) in its body one or more times. [Chapter 4](#) presents the `do...while` and `for` statements. [Chapter 6](#) presents the range-based `for` statement.

## Keywords

Each of the words `if`, `else`, `switch`, `while`, `do` and `for` are C++ keywords. Keywords cannot be used as identifiers, such as variable names, and must contain only lowercase letters. The following table shows the complete list of C++ keywords:

## C++ Keywords

*Keywords common to the C and C++ programming languages*

asm	auto	break	case	char
const	continue	default	do	double
else	enum	extern	float	for
goto	if	inline	int	long
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile		while

*C++-only keywords*

alignas	alignof	and	and_eq	bitand
bitor	bool	catch	char16_t	char32_t
class	compl	const_cast	constexpr	decltype
delete	dynamic_cast	explicit	export	false
friend	mutable	namespace	new	noexcept
not	not_eq	nullptr	operator	or
or_eq	private	protected	public	reinterpret_cast
static_assert	static_cast	template	this	thread_local
throw	true	try	typeid	typename
using	virtual	wchar_t	xor	xor_eq

20

*C++20 keywords*

char8_t	concept	char16_t	constexpr	constinit
co_await	co_return	co_yield	requires	

## Other Special Identifiers

**20** The C++20 standard indicates that the following identifiers should not be used in your code because they have special meanings in some contexts or are identifiers that may become keywords in the future:

- Non-keyword identifiers with special meaning—`final`, `import`, `module`, `override`, `transaction_safe` and `transaction_safe_dynamic`.
- Experimental keywords—`atomic_cancel`, `atomic_commit`, `atomic_noexcept`, `refexpr` and `synchronized`.

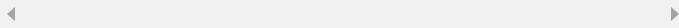
### 3.2.4 Summary of Control Statements

C++ has only three kinds of control structures, which from this point forward, we refer to as control statements—the sequence statement, selection statements (three types) and iteration statements (four types). Every program is formed by combining these statements as appropriate for the algorithm the program implements. We can model each control statement as an activity diagram. Each diagram contains an initial state and a final state that represent a control statement’s entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build programs—we simply connect the exit point of one to the entry point of the next using **control-statement stacking**. There’s only one other way in which you may connect control statements—**control-statement nesting** in which one control statement appears inside another. Thus, algorithms in C++ programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.

## 3.3 **IF** SINGLE-SELECTION STATEMENT

We introduced the **if** single-selection statement briefly in Section 2.6. Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The C++ statement

```
if (studentGrade >= 60) {  
    cout << "Passed";  
}
```



determines whether the condition `studentGrade >= 60` is true. If so, "Passed" is printed, and the next statement in order is performed. If the condition is false, the output statement is ignored, and the next statement in order is performed. The indentation of the second line of this selection statement is optional, but recommended for program clarity.

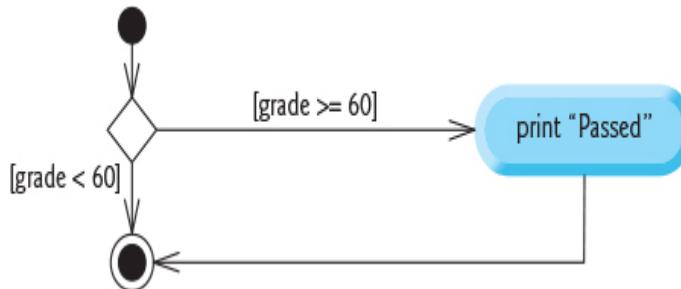
### bool Data Type

In Chapter 2, you created conditions using the relational or equality operators. Actually, any expression that evaluates to zero or nonzero can be used as a condition. Zero is treated as false, and nonzero is treated as true. C++ also provides the data type **bool** for Boolean variables that can hold only the values **true** and **false**—each of these is a C++ keyword.

For compatibility with earlier versions of C, which used integers for Boolean values, the `bool` value `true` also can be represented by any nonzero value (compilers typically use 1), and the `bool` value `false` also can be represented as zero.

### UML Activity Diagram for an `if` Statement

The following diagram illustrates the single-selection `if` statement.



This figure contains the most important symbol in an activity diagram—the diamond, or **decision symbol**, which indicates that a decision is to be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging

from a decision symbol has a guard condition (specified in square brackets next to the arrow). If a guard condition is true, the workflow enters the action state to which the transition arrow points. The diagram shows that if the grade is greater than or equal to 60 (i.e., the condition is true), the program prints “Passed,” then transitions to the activity’s final state. If the grade is less than 60 (i.e., the condition is false), the program immediately transitions to the final state without displaying a message. The `if` statement is a single-entry/single-exit control statement.

### 3.4 IF...ELSE DOUBLE-SELECTION STATEMENT

The `if` single-selection statement performs an indicated action only when the condition is true. The **if...else double-selection statement** allows you to specify an action to perform when the condition is true and another action when the condition is false. For example, the C++ statement

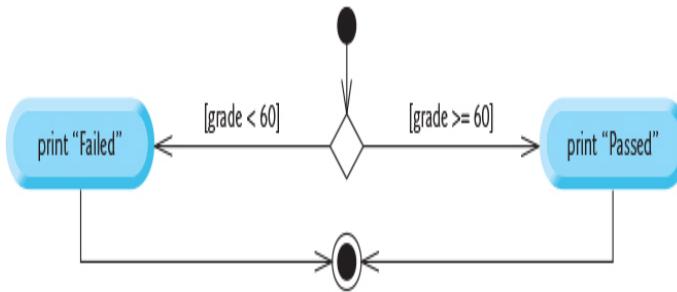
```
if (grade >= 60) {  
    cout << "Passed";  
}  
else {  
    cout << "Failed";  
}
```

prints “Passed” if `grade >= 60`, but prints “Failed” if it’s less than 60. In either case, after printing occurs, the next statement in sequence is performed.

The body of the `else` is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs.

#### UML Activity Diagram for an if...else Statement

The following diagram illustrates the flow of control in the preceding `if...else` statement:



### 3.4.1 Nested if...else Statements

A program can test multiple cases by placing `if...else` statements inside other `if...else` statements to create **nested if...else statements**. For example, the following nested `if...else` prints "A" for exam grades greater than or equal to 90, "B" for grades 80 to 89, "C" for grades 70 to 79, "D" for grades 60 to 69 and "F" for all other grades. We use shading to highlight the nesting.

```

if (studentGrade >= 90) {
    cout << "A";
}
else {
    if (studentGrade >= 80) {
        cout << "B";
    }
    else {
        if (studentGrade >= 70) {
            cout << "C";
        }
        else {
            if (studentGrade >= 60) {
                cout << "D";
            }
            else {
                cout << "F";
            }
        }
    }
}

```

If variable `studentGrade` is greater than or equal to 90, the first four conditions in the nested `if...else` statement will be true, but only the statement in the `if`-part of the first `if...else` statement will execute. After that statement executes, the `else`-part of the “outermost” `if...else` statement is skipped. The preceding nested `if...else` statement also can be written in the following form, which is identical except for the spacing and indentation that the compiler ignores:

---

```
if (studentGrade >= 90) {  
    cout << "A";  
}  
else if (studentGrade >= 80) {  
    cout << "B";  
}  
else if (studentGrade >= 70) {  
    cout << "C";  
}  
else if (studentGrade >= 60) {  
    cout << "D";  
}  
else {  
    cout << "F";  
}
```



This form avoids deep indentation of the code to the right, which can force lines to wrap. Throughout the text, we always enclose control statement bodies in braces (`{` and `}`), which avoids a logic error called the “dangling-`else`” problem.

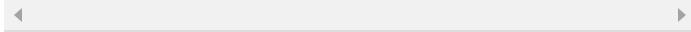
### 3.4.2 Blocks

The `if` statement expects only one statement in its body. To include several statements in its body (or the body of an `else` for an `if...else` statement), enclose the statements in braces. It’s good practice always to use the braces. Statements contained in a pair of braces (such as the body of a control statement or function) form a **block**. A block can be placed anywhere in a function that a single statement can be placed.

The following example includes a block of multiple statements in the `else` part of an `if...else` statement:

---

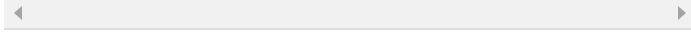
```
if (grade >= 60) {  
    cout << "Passed";  
}  
else  
{  
    cout << "Failed\n";  
    cout << "You must retake this course.";  
}
```



In this case, if `grade` is less than 60, the program executes both statements in the body of the `else` and prints

---

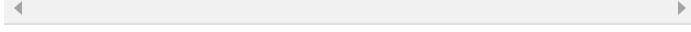
```
Failed  
You must retake this course.
```



Without the braces surrounding the two statements in the `else` clause, the statement

---

```
cout << "You must retake this course.;"
```



would be outside the body of the `else` part of the `if...else` statement and would execute regardless of whether the `grade` was less than 60—a logic error.

### Empty Statement

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an **empty statement**, which is represented by placing a semicolon (`;`) where a statement typically would be.

#### 3.4.3 Conditional Operator (`? :`)

C++ provides the **conditional operator** (`? :`) that can be used in place of an `if...else` statement. This can make your code shorter and clearer. The conditional operator is C++'s only **ternary operator** (i.e., an operator that takes three operands). Together, the operands and the `? :` symbol form a **conditional expression**. For example, the statement

---

```
cout << (studentGrade >= 60 ? "Passed" : "Failed"
```



prints the value of the conditional expression. The operand to the left of the ? is a condition. The second operand (between the ? and :) is the value of the conditional expression if the condition is true. The operand to the right of the : is the value of the conditional expression if the condition is false. The conditional expression in this statement evaluates to the string "Passed" if the condition

```
studentGrade >= 60
```

is true and to the string "Failed" if it's false. Thus, this statement with the conditional operator performs essentially the same function as the first `if...else` statement in Section 3.4. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses.

The values in a conditional expression also can be actions to execute. For example, the following conditional expression also prints "Passed" or "Failed":

```
grade >= 60 ? cout << "Passed" : cout << "Failed"
```

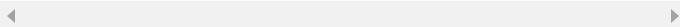
The preceding is read, "If `grade` is greater than or equal to 60, then `cout << "Passed"`; otherwise, `cout << "Failed"`." This is comparable to an `if...else` statement. Conditional expressions can appear in some program locations where `if...else` statements cannot.

### 3.5 WHILE ITERATION STATEMENT

An iteration statement allows you to specify that a program should repeat an action while some condition remains true.

As an example of C++'s **while iteration statement**, consider a program segment that finds the first power of 3 larger than 100. After the following `while` statement executes, the variable `product` contains the result:

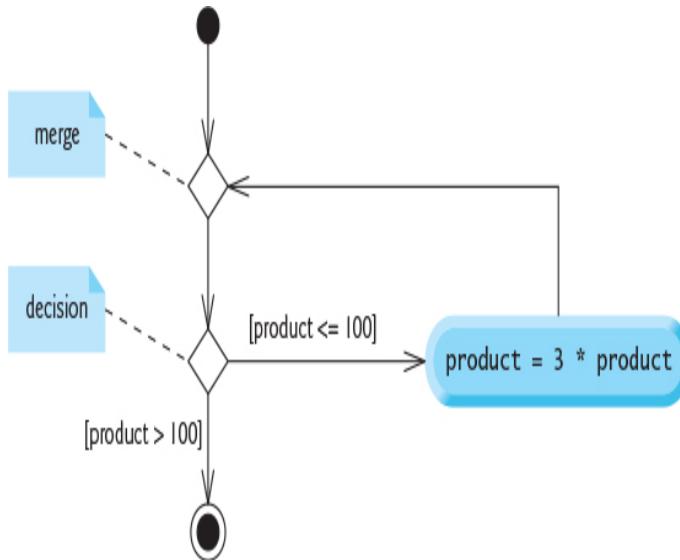
```
int product{3};  
  
while (product <= 100) {  
    product = 3 * product;  
}
```



Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, `product <= 100` becomes false. This terminates the iteration, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

### UML Activity Diagram for a `while` Statement

The UML activity diagram for the preceding `while` statement introduces the UML's **merge symbol**:



The UML represents both the merge symbol and the decision symbol as diamonds. The merge symbol joins two flows of activity into one. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing.

You can distinguish the decision and merge symbols by the number of incoming and outgoing transition arrows. A decision symbol has one transition arrow pointing to the

diamond and two or more pointing out from it to indicate possible transitions from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

## 3.6 COUNTER-CONTROLLED ITERATION

Consider the following problem statement:

*A class of ten students took a quiz. The grades (integers in the range 0–100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The program must input each grade, keep track of the total of all grades entered, perform the averaging calculation and print the result.

We use **counter-controlled iteration** to input the grades one at a time. This technique uses a counter to control the number of times a set of statements will execute. In this example, iteration terminates when the counter exceeds 10.

### 3.6.1 Implementing Counter-Controlled Iteration

In Fig. 3.1, the `main` function implements the class-averaging algorithm with counter-controlled iteration. It allows the user to enter 10 grades, then calculates and displays the average.

---

```
1 fig03_01.cpp
2 // Solving the class-average problem using counter-controlled iteration.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     // initialization phase
8     int total{0}; // initialize sum of grades entered by the user
9     int gradeCounter{1}; // initialize grade # to be entered next
10
11    // processing phase uses counter-controlled iteration
12    while (gradeCounter <= 10) { // loop 10 times
13        cout << "Enter grade: "; // prompt
14        int grade;
15        cin >> grade; // input next grade
16        total = total + grade; // add grade to total
17        gradeCounter = gradeCounter + 1; // increment counter by 1
18    }
19
20    // termination phase
21    int average{total / 10}; // int division yields int result
```

```
22  
23 // display total and average of grades  
24 cout << "\nTotal of all 10 grades is " << total;  
25 cout << "\nClass average is " << average << endl;  
26 }
```

```
Enter grade: 67  
Enter grade: 78  
Enter grade: 89  
Enter grade: 67  
Enter grade: 87  
Enter grade: 98  
Enter grade: 93  
Enter grade: 85  
Enter grade: 82  
Enter grade: 100
```

```
Total of all 10 grades is 846  
Class average is 84
```

**Fig. 3.1** Solving the class-average problem using counter-controlled iteration. (Part 1 of 2.)

### Local Variables in `main`

Lines 8, 9, 14 and 21 declare `int` variables `total`, `gradeCounter`, `grade` and `average`, respectively.

Variable `grade` stores the user input. A variable declared in a function body is a local variable. It can be used only from the line of its declaration to the closing right brace of the block in which the variable is declared. A local variable's declaration must appear before the variable is used. Variable `grade`—declared in the body of the `while` loop—can be used only in that block.

### Initializing Variables `total` and `gradeCounter`

Lines 8–9 declare and initialize `total` to 0 and `gradeCounter` to 1. These initializations occur before the variables are used in calculations.

### Reading 10 Grades from the User

The `while` statement (lines 12–18) continues iterating as long as `gradeCounter`'s value is less than or equal to 10. Line 13 displays the prompt "Enter grade: ". Line 15 inputs the grade entered by the user and assigns it to variable `grade`. Then line 16 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, replacing its previous value. Line 17 adds 1 to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes it to exceed 10, which terminates the loop.

### Calculating and Displaying the Class Average

When the loop terminates, line 21 performs the averaging calculation in the `average` variable's initializer. Line 24 displays the text "Total of all 10 grades is " followed by variable `total`'s value. Then, line 25 displays the text "Class average is " followed by `average`'s value. When execution reaches line 26, the program terminates.

### 3.6.2 Integer Division and Truncation

This example's averaging calculation produces an integer result. The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield 84.6. Numbers like 84.6 that contain decimal points are **floating-point numbers**. In the class-average program, however, the result of `total / 10` is the integer 84, because `total` and `10` are both integers.

Dividing two integers results in **integer division**—any fractional part of the calculation is truncated. In the next section, we'll see how to obtain a floating-point result from the averaging calculation.

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example,  $7 / 4$ , which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

## 3.7 SENTINEL-CONTROLLED ITERATION

Let's generalize Section 3.6's class-average problem. Consider the following problem:

*Develop a class-averaging program that processes grades for an arbitrary number of students each time it's run.*

In the previous class-average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. The program must process an *arbitrary* number of grades.

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." The user enters grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that no more grades will be entered.

You must choose a sentinel value that cannot be confused with an acceptable input value. Grades on a quiz are nonnegative integers, so  $-1$  is an acceptable sentinel value for this problem. Thus, a run of the class-averaging program might process a stream of inputs such as 95, 96, 75, 74, 89 and  $-1$ . The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89; since  $-1$  is the sentinel value, it should not enter into the averaging calculation.

It's possible that the user could enter  $-1$  before entering grades, in which case the number of grades will be zero. We must test for this case before calculating the class average. According to the C++ standard, the result of division by zero in floating-point arithmetic is undefined. When performing division (/) or remainder (%) calculations in which the right operand could be zero, test for this and handle it (e.g., display an error message) rather than allowing the calculation to proceed.

### 3.7.1 Implementing Sentinel-Controlled Iteration

In Fig. 3.2, the `main` function implements sentinel-controlled iteration. Although each grade entered by the user is an integer, the averaging calculation is likely to produce a floating-point number. The type `int` cannot represent such a number. C++ provides data types `float` and `double` to store floating-point numbers in memory. The primary difference between these types is that `double` variables typically store numbers with larger magnitude and finer detail—that is, more digits to the right of the decimal point, which is also known as the number’s `precision`. C++ also supports type `long double` for floating-point values with larger magnitude and more precision than `double`. We say more about floating-point types in Chapter 4.

---

```
1 // fig03_02.cpp
2 // Solving the class-average problem using sentinel-controlled iteration.
3 #include <iostream>
4 #include <iomanip> // parameterized stream manipulators
5 using namespace std;
6
7 int main() {
8     // initialization phase
9     int total{0}; // initialize sum of grades
10    int gradeCounter{0}; // initialize # of grades entered so far
11
12    // processing phase
13    // prompt for input and read grade from user
14    cout << "Enter grade or -1 to quit: ";
15    int grade;
16    cin >> grade;
17
18    // loop until sentinel value is read from user
19    while (grade != -1) {
20        total = total + grade; // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22
23        // prompt for input and read next grade from user
24        cout << "Enter grade or -1 to quit: ";
25        cin >> grade;
26    }
27
28    // termination phase
29    // if user entered at least one grade...
30    if (gradeCounter != 0) {
31        // use number with decimal point to calculate average of grades
32        double average{static_cast<double>(total) / gradeCounter};
33
34        // display total and average (with two digits of precision)
35        cout << "\nTotal of the " << gradeCounter
36            << " grades entered is " << total;
37        cout << setprecision(2) << fixed;
38        cout << "\nClass average is " << average << endl;
39    }
40    else { // no grades were entered, so output appropriate message
41        cout << "No grades were entered" << endl;
```

```
42 }  
43 }
```

```
Enter grade or -1 to quit: 97  
Enter grade or -1 to quit: 88  
Enter grade or -1 to quit: 72  
Enter grade or -1 to quit: -1
```

```
Total of the 3 grades entered is 257  
Class average is 85.67
```

**Fig. 3.2** Solving the class-average problem using sentinel-controlled iteration. (Part 1 of 2.)

Recall that integer division produces an integer result. This program introduces a **cast operator** to force the averaging calculation to produce a floating-point numeric result. This program also stacks control statements on top of one another (in sequence)—the **while** statement (lines 19–26) is followed in sequence by an **if...else** statement (lines 30–42). Much of the code in this program is identical to that in Fig. 3.1, so we concentrate on only the new concepts.

### Program Logic for Sentinel-Controlled Iteration vs. Counter-Controlled Iteration

Line 10 initializes **gradeCounter** to 0 because no grades have been entered yet. Remember that this program uses sentinel-controlled iteration to input the grades. The program increments **gradeCounter** only when the user enters a valid grade. Line 32 declares double variable **average**, which stores the calculated class average as a floating-point number.

Compare the program logic for sentinel-controlled iteration in this program with that for counter-controlled iteration in Fig. 3.1. In counter-controlled iteration, each iteration of the **while** statement (lines 12–18 of Fig. 3.1) reads a value from the user, for the specified number of iterations. In sentinel-controlled iteration, the program prompts for and reads the first value (lines 14 and 16 of Fig. 3.2) before reaching the **while**. This value determines whether the flow of control

should enter the `while`'s body. If the condition is false, the user entered the sentinel value, so no grades were entered and the body does not execute. If the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` and increments the `gradeCounter`. Then lines 24–25 in the loop body input the next value from the user. Next, program control reaches the closing right brace of the loop at line 26, so execution continues with the test of the `while`'s condition (line 19). The condition uses the most recent `grade` entered by the user to determine whether the loop body should execute again.

The next `grade` is always input from the user immediately before the `while` condition is tested. This allows the program to determine whether the value just input is the sentinel value before the program processes that value (i.e., adds it to the `total`). If the sentinel value is input, the loop terminates, and the program does not add `-1` to the `total`.

After the loop terminates, the `if...else` statement at lines 30–42 executes. The condition at line 30 determines whether any grades were input. If none were input, the `if...else` statement's `else` part executes and displays the message "No grades were entered". After the `if...else` executes, the program terminates.

### 3.7.2 Converting Between Fundamental Types Explicitly and Implicitly

If at least one grade was entered, line 32 of Fig. 3.2

---

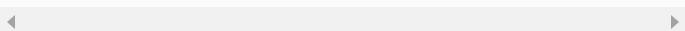
```
double average{static_cast<double>(total) / grad
```



calculates the average. Recall from Fig. 3.1 that integer division yields an integer result. Even though variable `average` is declared as a `double`, if we had written line 32 as

---

```
double average{total / gradeCounter};
```



it would lose the fractional part of the quotient before the result of the division was used to initialize `average`.

### `static_cast` Operator

To perform a floating-point calculation with integers in this example, you first create temporary floating-point values using the **`static_cast` operator**. Line 32 converts a temporary copy of its operand in parentheses (`total`) to the type in angle brackets (`double`). The value stored in the original `int` variable `total` is still an integer. Using a cast operator in this manner is called **explicit conversion**. `static_cast` is one of several cast operators we'll discuss.

### Promotions

After the cast operation, the calculation consists of the temporary `double` copy of `total` divided by the integer `gradeCounter`. For arithmetic, the compiler knows how to evaluate only expressions in which all the operand types are identical. To ensure this, the compiler performs an operation called **promotion** (also called **implicit conversion**) on selected operands. In an expression containing values of data types `int` and `double`, C++ **promotes** `int` operands to `double` values. So in line 32, C++ promotes a temporary copy of `gradeCounter`'s value to type `double`, then performs the division. Finally, `average` is initialized with the floating-point result. Section 5.5 discusses the allowed fundamental-type promotions.

### Cast Operators for Any Type

Cast operators are available for use with every fundamental type and for other types, as you'll see beginning in [Chapter 9](#). Simply specify the type in the angle brackets (< and >) that follow the `static_cast` keyword. It's a **unary operator**—that is, it has only one operand. Other unary operators include the unary plus (+) and minus (-) operators for expressions such as `- 7` or `+5`. Cast operators have the second highest precedence.

### 3.7.3 Formatting Floating-Point Numbers

The formatting capabilities in Fig. 3.2 are introduced here briefly and explained in depth in Chapter 15.

#### `setprecision` Parameterized Stream Manipulator

Line 37's call to `setprecision`—`setprecision(2)`—indicates that floating-point values should be output with *two* digits of **precision** to the right of the decimal point (e.g., 92.37). `setprecision` is a **parameterized stream manipulator** because it requires an argument (in this case, 2) to perform its task. Programs that use parameterized stream manipulators must include the header `<iomanip>`. The manipulator `endl` (lines 38 and 41) from `<iostream>` is a **nonparameterized stream manipulator** because it does not require an argument.

#### `fixed` Nonparameterized Stream Manipulator

The stream manipulator `fixed` (line 37) indicates that floating-point values should be output in **fixed-point format**. This is as opposed to **scientific notation**<sup>4</sup>, which displays a number between the values of 1.0 and 10.0, multiplied by a power of 10. So, in scientific notation, the value 3,100.0 is displayed as  $3.1 \times 10^3$  (that is,  $3.1 \times 10^3$ ). This format is useful for displaying very large or very small values.

4. Formatting using scientific notation is discussed further in Chapter 15.

Fixed-point formatting forces a floating-point number to display without scientific notation. Fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole-number amount, such as 88.00. Without the fixed-point formatting option, 88.00 prints as 88 without the trailing zeros and decimal point.

The stream manipulators `setprecision` and `fixed` perform **sticky settings**. Once they're specified, all floating-point values formatted in your program will use those settings until you change them. Chapter 15 shows how to capture the stream format settings before applying sticky settings, so you can restore the original format settings later.

## Rounding Floating-Point Numbers

When the stream manipulators `fixed` and `setprecision` are used, the printed value is **rounded** to the number of decimal positions specified by the current precision. The value in memory remains unaltered. For a precision of 2, the values 87.946 and 67.543 are rounded to 87.95 and 67.54, respectively.<sup>5</sup>

5. In Fig. 3.2, if you do not specify `setprecision` and `fixed`, C++ uses four digits of precision by default. If you specify only `setprecision`, C++ uses six digits of precision.

Together, lines 37 and 38 of Fig. 3.2 output the class average rounded to the nearest hundredth and with exactly two digits to the right of the decimal point. The three grades entered during the execution of the program in Fig. 3.2 total 257, which yields the average 85.666... and displays the rounded value 85.67.

## 3.8 NESTED CONTROL STATEMENTS

We've seen that control statements can be stacked on top of one another (in sequence). In this case study, we examine the only other structured way control statements can be connected—namely, by **nesting** one control statement within another.

### 3.8.1 Problem Statement

Consider the following problem statement:

*A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.*

*Your program should analyze the results of the exam as follows:*

1. Input each test result (i.e., a 1 or a 2).  
Display the message “Enter result” on the screen each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results, indicating the number of students who passed and the number who failed.
4. If more than eight students passed the exam, print “Bonus to instructor!”

### 3.8.2 Implementing the Program

Figure 3.3 implements the program with counter-controlled iteration and shows two sample executions. Lines 8–10 and 16 of `main` declare the variables that are used to process the examination results.

---

```
1 // fig03_03.cpp
2 // Analysis of examination results using nested control statements.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     // initializing variables in declarations
8     int passes{0};
9     int failures{0};
10    int studentCounter{1};
11
12    // process 10 students using counter-controlled loop
13    while (studentCounter <= 10) {
14        // prompt user for input and obtain value from user
15        cout << "Enter result (1 = pass, 2 = fail): ";
16        int result;
17        cin >> result;
18
19        // if...else is nested in the while statement
20        if (result == 1) {
21            passes = passes + 1;
22        }
23        else {
24            failures = failures + 1;
25        }
26
27        // increment studentCounter so loop eventually terminates
28        studentCounter = studentCounter + 1;
29    }
30
31    // termination phase; prepare and display results
32    cout << "Passed: " << passes << "\nFailed: " << failures << endl;
33
34    // determine whether more than 8 students passed
35    if (passes > 8) {
36        cout << "Bonus to instructor!" << endl;
37    }
38 }
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

**Fig. 3.3** Analysis of examination results using nested control statements. (Part 1 of 2.)

The `while` statement (lines 13–29) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 20–25) for processing each result is nested in the `while` statement. If the `result` is 1, the `if...else` statement increments `passes`; otherwise, it assumes the `result` is 2<sup>6</sup> and increments `failures`. Line 28 increments `studentCounter` before the loop condition is tested again at line 13. After 10 values have been input, the loop terminates and line 32 displays the

number of passes and failures. The `if` statement at lines 35–37 determines whether more than eight students passed the exam and, if so, outputs the message "Bonus to instructor!"<sup>6</sup>

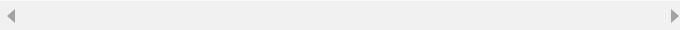
<sup>6</sup> This could be a bad assumption if invalid data is entered. We'll discuss data validation techniques later.

Figure 3.3 shows the input and output from two sample executions. During the first, the condition at line 35 is true—more than eight students passed the exam, so the program outputs a message to bonus the instructor.

### 11 3.8.3 Preventing Narrowing Conversions with C++11 List Initialization

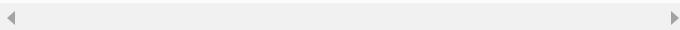
Consider the C++11 list initialization in line 10 of Fig. 3.3:

```
int studentCounter{1};
```



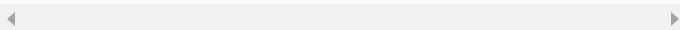
Prior to C++11, you would have written this as

```
int studentCounter = 1;
```



For fundamental-type variables, list-initialization syntax prevents **narrowing conversions** that could result in *data loss*. For example, the declaration

```
int x = 12.7;
```



attempts to assign the `double` value `12.7` to the `int` variable `x`. Here, C++ converts the `double` value to an `int` by truncating the floating-point part (`.7`). This is a narrowing conversion that loses data. So, this declaration assigns `12` to `x`. Compilers typically issue a warning for this, but still compile the code.

However, using list initialization, as in

```
int x{12.7};
```



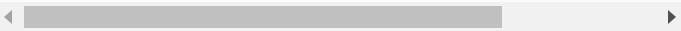
---

yields a *compilation error*, helping you avoid a potentially subtle logic error. If you specify a whole-number `double` value, like `12.0`, you'll still get a compilation error. The initial-izer's type (`double`), not its value (`12.0`), determines whether a compilation error occurs.

The C++ standard document does not specify the wording of error messages. For the preceding declaration, Apple's Xcode compiler gives the error

---

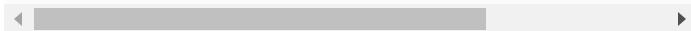
```
Type 'double' cannot be narrowed to 'int' in ini
```



Visual Studio gives the error

---

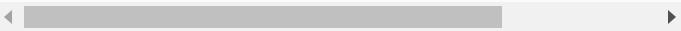
```
conversion from 'double' to 'int' requires a nar
```



and GNU C++ gives the error

---

```
type 'double' cannot be narrowed to 'int' in ini  
[-Wc++11-narrowing]
```



We'll discuss additional list-initializer features in later chapters.

### A Look Back at Fig. 3.1

You might think that the following statement from Fig. 3.1

---

```
int average{total / 10}; // int division yields
```

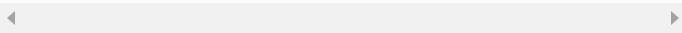


contains a narrowing conversion, but `total` and `10` are both `int` values, so the initializer value is an `int`. If in the preceding statement `total` were a `double` variable or if we used the `double` literal value `10.0` for the denominator, then the initializer value would have type `double` and the compiler would issue an error message for a narrowing conversion.

## 3.9 COMPOUND ASSIGNMENT OPERATORS

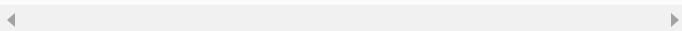
You can abbreviate the statement

```
c = c + 3;
```



with the **addition compound assignment operator**, **`+=`**, as

```
c += 3;
```



The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left. Thus, the assignment expression `c += 3` adds 3 to `c`. The following table shows all the arithmetic compound assignment operators, sample expressions and explanations of what the operators do:

Operator	Sample expression	Explanation	Assigns
<i>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to <code>c</code>
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to <code>d</code>
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to <code>e</code>
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 to <code>f</code>
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 to <code>g</code>

Later, we'll discuss other types of compound assignment operators.

## 3.10 INCREMENT AND DECREMENT OPERATORS

The following table summarizes C++'s two unary operators for adding 1 to or subtracting 1 from the value of a numeric

variable—these are the unary **increment operator**, **`++`**, and the unary **decrement operator**, **`--`**:

Operator	Operator name	Sample expression	Explanation
<code>++</code>	prefix increment	<code>++number</code>	Increment number by 1, then use the new value of number in the expression in which number resides.
<code>++</code>	postfix increment	<code>number++</code>	Use the current value of number in the expression in which number resides, then increment number by 1.
<code>--</code>	prefix decrement	<code>--number</code>	Decrement number by 1, then use the new value of number in the expression in which number resides.
<code>--</code>	postfix decrement	<code>number--</code>	Use the current value of number in the expression in which number resides, then decrement number by 1.

An increment or decrement operator that's prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

Using the prefix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable **preincrements** (or **predecrements**) the variable. The variable is incremented (or decremented) by 1 then its new value is used in the expression in which it appears.

Using the postfix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable **postincrements** (or **postdecrements**) the variable. The variable's current value is used in the expression in which it appears then its value is incremented (or decremented) by 1. Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

## Prefix Increment vs. Postfix Increment

Figure 3.4 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator. The decrement operator (`- -`) works similarly.

```
1 // fig03_04.cpp
2 // Prefix increment and postfix increment operators.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     // demonstrate postfix increment operator
8     int c{5};
9     cout << "c before postincrement: " << c << endl; // prints 5
10    cout << "    postincrementing c: " << c++ << endl; // prints 5
11    cout << "c after postincrement: " << c << endl; // prints 6
12
13    cout << endl; // skip a line
14
15    // demonstrate prefix increment operator
16    C = 5;
17    cout << "c before preincrement: " << c << endl; // prints 5
18    cout << "    preincrementing c: " << ++c << endl; // prints 6
19    cout << "c after preincrement: " << c << endl; // prints 6
20 }
```

```
c before postincrement: 5
postincrementing c: 5
c after postincrement: 6
```

```
c before preincrement: 5
preincrementing c: 6
c after preincrement: 6
```

**Fig. 3.4** Prefix increment and postfix increment operators.

Line 8 initializes the variable `C` to 5, and line 9 outputs `C`'s initial value. Line 10 outputs the value of the expression `c++`. This expression postincrements the variable `c`, so `c`'s original

value (5) is output, then C's value is incremented (to 6). Thus, line 10 outputs C's initial value (5) again. Line 11 outputs C's new value (6) to prove that the variable's value was indeed incremented in line 10.

Line 16 resets C's value to 5, and line 17 outputs C's value. Line 18 outputs the value of the expression `++C`. This expression preincrements C, so its value is incremented; then the new value (6) is output. Line 19 outputs C's value again to show that the value of C is still 6 after line 18 executes.

### Simplifying Statements with the Arithmetic Compound Assignment, Increment and Decrement Operators

The arithmetic compound assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 3.3 (lines 21, 24 and 28)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with compound assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect. Only when a variable appears in the context of a larger expression does preincrementing or postincrementing the variable have a different effect (and similarly for predecrementing or postdecrementing).

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not a variable.

## Operator Precedence and Grouping

The following table shows the precedence and grouping of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column indicates the grouping of the operators at each level of precedence. Notice that the conditional operator (`? :`), the unary operators preincrement (`++`), predecrement (`--`), plus (`+`) and minus (`-`), and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` group from *right-to-left*. All other operators in this table group from *left-to-right*. The third column names the various groups of operators.

Operators	Grouping	Type
<code>:: ()</code>	left to right <i>[See Chapter 2's caution regarding grouping parentheses.]</i>	primary
<code>++ -- static_cast&lt;type&gt;()</code>	left to right	postfix
<code>++ -- + -</code>	right to left	unary (prefix)
<code>* / %</code>	left to right	multiplicative
<code>+ -</code>	left to right	additive
<code>&lt;&lt; &gt;&gt;</code>	left to right	insertion/extraction
<code>&lt; &lt;= &gt; &gt;=</code>	left to right	relational
<code>== !=</code>	left to right	equality
<code>?:</code>	right to left	conditional
<code>= += -= *= /= %=</code>	right to left	assignment

### 3.11 FUNDAMENTAL TYPES ARE NOT PORTABLE

You can view the complete list of C++ fundamental types and their typical ranges at

<https://en.cppreference.com/w/cpp/language/types>

In C and C++, an `int` on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes), and on another machine by 64 bits (8 bytes). For this reason, code using integers is not always portable across platforms. You could write multiple versions of your programs to use different integer types on different platforms. Or you could use techniques to achieve various levels of portability.<sup>7</sup> In the next section, we'll show one way to achieve portability.

<sup>7</sup>. The integer types in the header `<cstdint>` (<https://en.cppreference.com/w/cpp/types/integer>) can

be used to ensure that integer variables are correct size for your application across platforms.

**PERF** Among C++’s integer types are `int`, `long` and `long long`. The C++ standard requires type `int` to be at least 16 bits, type `long` to be at least 32 bits and type `long long` to be at least 64 bits. The standard also requires that an `int`’s size be less than or equal to a `long`’s size and that a `long`’s size be less than or equal to a `long long`’s size. Such “squishy” requirements create portability challenges, but allow compiler implementers to optimize performance by matching fundamental types sizes to your machine’s hardware.

### 3.12 OBJECTS NATURAL CASE STUDY: ARBITRARY SIZED INTEGERS

The range of values an integer type supports depends on the number of bytes used to represent the type on a particular computer. For example, a four-byte `int` can store  $2^{32}$  possible values in the range –2,147,483,648 to 2,147,483,647. On most systems, a `long long` integer is 8 bytes and can store  $2^{64}$  possible values in the range –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

#### Some Applications Need Numbers Outside a `long long` Integer’s Range

Consider factorial calculations. A factorial is the product of the integers from 1 to a given value. The factorial of 5 (written 5!) is  $1 * 2 * 3 * 4 * 5$ , which is 120. The highest factorial value we can represent in a 64-bit integer is 20!, which is 2,432,902,008,176,640,000. Factorials quickly grow outside the range representable by a `long long` integer. With big data getting bigger quickly, an increasing number of real-world applications will exceed the limitations of `long long` integers.



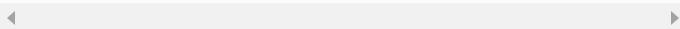
**SECURITY** Another application requiring extremely large integers is cryptography—an important aspect of securing data that’s transmitted between computers over the Internet. Many cryptography algorithms perform calculations using 128-bit or

256-bit integer values—far larger than we can represent with C++’s fundamental types.

### Arbitrary Precision Integers with Class `BigNumber`

Any application requiring integers outside `long long`’s range requires special processing. Unfortunately, the C++ standard library does not (yet) have a class for arbitrary precision integers. So, for this example, we’ll dive into the vast world of open-source class libraries to demonstrate one of the many C++ classes that you can use to can create and manipulate arbitrary precision integers. We’ll use the class **BigNumber** from:

<https://github.com/limeoats/BigNumber>



For your convenience, we included the download in this example’s `fig03_05` folder. Be sure to read the license terms included in the provided `LICENSE.md` file.

To use `BigNumber`, you don’t have to understand how it’s implemented.<sup>8</sup> You simply include its header file (`bignum.h`), create objects of the class then use them in your code. Figure 3.5 demonstrates `BigNumber` and shows a sample output. For this example, we’ll use the maximum `long long` integer value to show that we can create an even bigger integer with `BigNumber`. At the end of this section, we show how to compile and run the code.

<sup>8</sup>. After you get deeper into C++, you might want to peek at `BigNumber`’s source code (approximately 1000 lines) to see how it’s implemented. In our object-oriented programming presentation later in this book, you’ll learn a variety of techniques that you can use to create your own big integer class.

---

```
1 // fig03_05.cpp
2 // Integer ranges and arbitrary precision integers.
3 #include <iostream>
4 #include "bignumber.h"
5 using namespace std;
6
7 int main() {
8     // use the maximum long long fundamental type value in calculations
9     long long value1{9'223'372'036'854'775'807LL}; // max long long value
10    cout << "long long value1: " << value1
11    << "\nvalue1 - 1 = " << value1 - 1 // OK
12    << "\nvalue1 + 1 = " << value1 + 1; // result is undefined
13
14    // use an arbitrary precision integer
15    BigNumber value2{value1};
16    cout << "\n\nBigNumber value2: " << value2
17    << "\nvalue2 - 1 = " << value2 - 1 // OK
18    << "\nvalue2 + 1 = " << value2 + 1; // OK
19
20    // powers of 100,000,000 with long long
21    long long value3{100'000'000};
22    cout << "\n\nvalue3: " << value3;
23
24    int counter{2};
25
26    while (counter <= 5) {
27        value3 *= 100'000'000; // quickly exceeds maximum long long value
28        cout << "\nvalue3 to the power " << counter << ":" << value3;
29        ++counter;
30    }
}
```

```
31
32 // powers of 100,000,000 with BigNumber
33 BigNumber value4{100'000'000};
34 cout << "\n\nvalue4: " << value4 << endl;
35
36 counter = 2;
37
38 while (counter <= 5) {
39     cout << "value4.pow(" << counter << ")";
40     << value4.pow(counter) << endl;
41     ++counter;
42 }
43
44 cout << endl;
45 }
```

## **Fig. 3.5** Integer ranges and arbitrary precision integers. (Part 1 of 2.)

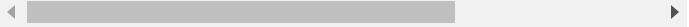
## Including a Header That Is Not in the C++ Standard Library

In an `#include` directive, headers that are not from the C++ Standard Library typically are placed in double quotes (" "), rather than the angle brackets (<>). The double quotes tell the compiler that header is in your application's folder or another folder that you specify.

### What Happens When You Exceed the Maximum `long long` Integer Value?

Line 9 initializes the variable `value1` with the maximum `long long` value on our system:<sup>9</sup>

```
long long value1{9'223'372'036'854'775'807LL}; /
```



9. The platforms on which we tested this book's code each have as their maximum `long long` integer value 9,223,372,036,854,775,807. You can determine this value programmatically with the expression `std::numeric_limits<long long>::max()`, which uses class `numeric_limits` from the C++ standard library header `<limits>`. The <> and :: notations used in this expression are covered in later chapters, so we used the literal value 9,223,372,036,854,775,807 in Fig. 3.5.

**20** Typing numeric literals with many digits can be error prone. To make such literals more readable and reduce errors, C++14 introduced the **digit separator** ' (a single-quote character), which you insert between groups of digits in numeric literals—we used it to separate groups of three digits. Also, note the **LL** (“el el”) at the end of the literal value—this indicates that the literal is a `long long` integer.

Line 10 displays `value1`, then line 11 subtracts one from it to demonstrate a valid calculation. Next, we attempt to add `1` to `value1`, which already contains the maximum `long long` value. All our compilers displayed as the result the *minimum long long value*. The C++ standard actually says the result of this calculation is **undefined behavior**. Such behaviors can differ between systems—ours displayed an incorrect value but other systems could terminate the program and display an error message. This is another example of why the fundamental integer types are not portable.

## Performing the Same Operations with a BigNumber Object

Lines 15–18 use a `BigNumber` object to repeat the operations from lines 9–12. We create a `BigNumber` object named `value2` and initialize it with `value1`, which contains the maximum value of a `long long` integer:

```
BigNumber value2{value1};
```

Next, we display the `BigNumber` then subtract one from it and display the result. Line 18 adds one to `value2`, which contains the maximum value of a `long long`. `BigNumber` handles arbitrary precision integers, so it *correctly performs this calculation*. The result is a value that C++’s fundamental integer types cannot handle on our systems.

`BigNumber` supports all the typical arithmetic operations, including `+` and `-` used in this program. The compiler already knows how to use arithmetic operators with fundamental numeric types, but it has to be taught how to handle those operators for class objects. We discuss that process—called operator overloading—in [Chapter 14](#).

## Powers of 100,000,000 with `long long` Integers

Lines 21–30 calculate powers of 100,000,000 using `long long` integers. First, we create the variable `value3` and display its value. Lines 26–30 loop five times. The calculation

```
value3 *= 100'000'000; // quickly exceeds maximum
```

multiples `value3`’s current value by 100,000,000 to raise `value3` to the next power. As you can see in the program’s output, only the loop’s first iteration produces a correct result.

## Powers of 100,000,000 with `BigNumber` Objects

To demonstrate that `BigNumber` can handle significantly larger values than the fundamental integer types, lines 33–42 calculate powers of 100,000,000 using a `BigNumber` object.

First, we create `BigNumber value4` and display its initial value. Lines 38–42 loop five times. The calculation

```
value4.pow(counter)
```

calls `BigNumber` member function `pow` to raise `value4` to the power `counter`. `BigNumber` correctly handles each calculation, producing massive values that are far outside the ranges supported by our Windows, macOS and Linux systems' fundamental integer types.



Though a `BigNumber` can represent any integer value, it does not match your system's hardware. So you'll likely sacrifice some performance in exchange for the flexibility `Big-Number` provides.

## Compiling and Running the Example in Microsoft Visual Studio

In Microsoft Visual Studio:

1. Create a new project, as described in Section 1.9.
2. In the **Solution Explorer**, right-click the project's **Source Files** folder and select **Add > Existing Item....**
3. Navigate to the `fig03_05` folder, select `fig03_05.cpp` and click **Add**.
4. Repeat Steps 2–3 for `bignumber.cpp` from the `fig03_05\BigNumber\src` folder.
5. In the **Solution Explorer**, right-click the project's name and select **Properties....**
6. Under **Configuration Properties**, select **C/C++**, then on the right side of the dialog, add to the **Additional Include Directories** the full path to the `BigNumber\src` folder on your system. For our system, this was  
`C:\Users\account\Documents\examples\ch03\fig03_05\BigNumber\src`
7. Click **OK**.

- 8.** Type *Ctrl + F5* to compile and run the program.

### Compiling and Running the Example in GNU g++

For GNU g++ (these instructions also work from a Terminal window on macOS):

- 1.** At your command line, change to this example's `fig03_05` folder.
- 2.** Type the following command to compile the program—the `-I` option specifies additional folders in which the compiler should search for header files:

---

```
g++ -std=c++2a -I BigNumber/src fig03_05.cpp \
BigNumber/src/bignum.cpp -o fig03_05
```

- 3.** Type the following command to execute the program:

---

```
./fig03_05
```

### Compiling and Running the Example in Apple Xcode

In Apple Xcode:

- 1.** Create a new project, as described in Section 1.9, and delete `main.cpp`.
- 2.** Drag `fig03_05.cpp` from the `fig03_05` folder in the Finder onto your project's source code folder in Xcode, then click **Finish** in the dialog that appears.
- 3.** Drag `bignum.h` and `bignum.cpp` from the `fig03_05/BigNumber/src` folder onto your project's source code folder in Xcode, then click **Finish** in the dialog that appears.
- 4.** Type  + `R` to compile and run the program.

## 3.13 C++20 FEATURE MOCK-UP—TEXT FORMATTING WITH FUNCTION FORMAT

**20** C++20 introduces powerful new string formatting capabilities via the **format function** (in header `<format>`).

These capabilities greatly simplify C++ formatting by using a

syntax similar to that used in Python, Microsoft’s .NET languages (like C# and Visual Basic) and the up-and-coming newer language Rust.<sup>10</sup> You’ll see throughout the book that the C++20 text-formatting capabilities are more concise and more powerful than those in earlier C++ versions. In Chapter 14, we’ll also show that these new capabilities can be customized to work with your own new class types. We’ll show both old- and new-style formatting because in your career you may work with software that uses the old style.

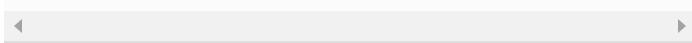
10. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.htm>

## C++20 String Formatting Is Not Yet Implemented

At the time of this writing (April 2020), the C++ standard was about to be approved (May 2020) and C++ compilers had not yet fully implemented many of C++20’s features, including text formatting. However, the `{fmt}` library at

---

<https://github.com/fmtlib/fmt>



provides a full implementation of the new text-formatting features.<sup>11</sup><sup>12</sup> So, we’ll use this library until the C++20 compilers implement text formatting.<sup>13</sup> For your convenience, we included the complete download in the `examples` folder’s `libraries` subfolder, then included only the required files in the `fig03_06` folder. Be sure to read the library’s license terms included in the provided `format.h` file.

11. According to <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.htm>, which is the C++ standard committee proposal for C++20 text formatting.

12. C++20’s text formatting features are a subset of the features provided by the `{fmt}` library.

13. Some of our C++20 Feature Mock-Up sections present code that does not compile or run. Once the compilers implement those features, we’ll retest the code, update our digital products and post updates for our print products at <https://deitel.com/c-plus-plus-20-for-programmers>. The code in this example runs, but uses the `{fmt}` open-source library to demonstrate features that C++20 compilers will support eventually.

## Format String Placeholders

The `format` function's first argument is a **format string** containing **placeholders** delimited by curly braces (`{` and `}`). The function replaces the placeholders with the values of the function's other arguments, as demonstrated in Fig. 3.6.

```
1 // fig03_06.cpp
2 // C++20 string formatting.
3 #include <iostream>
4 #include "fmt/format.h" // C++20: This will be #include <format>
5 using namespace std;
6 using namespace fmt; // not needed in C++20
7
8 int main() {
9     string student{"Paul"};
10    int grade{87};
11
12    cout << format("{}'s grade is {}", student, grade) << endl;
13 }
```

```
Paul's grade is 87
```

**Fig. 3.6** C++20 string formatting.

## Placeholders Are Replaced Left-to-Right

The `format` function replaces its format string argument's placeholders left-to-right by default. So, line 11's `format` call inserts into the format string

```
"{}'s grade is {}"
```

student's value ("Paul") in the first placeholder and grade's value (87) in the second placeholder, then returns the string

```
"Paul's grade is 87"
```

## Compiling and Running the Example in Microsoft Visual Studio

The steps below are the same as those in Section 3.12 with the following changes:

- In Step 3, navigate to the `fig03_06` folder, add both the files `fig03_06.cpp` and `format.cc` to your project's **Source Files** folder.
- In Step 6, add to the **Additional Include Directories** the full path to the `fig03_06` folder on your system. For our system, this was

```
C:\Users\account\Documents\examples\ch03\fig03_06
```

## Compiling and Running the Example in GNU g++

For GNU g++ (these instructions also work from a Terminal window on macOS):

1. At your command line, change folders to this example's `fig03_06` folder.
2. Type the following command to compile the program:

```
g++ -std=c++2a -I fmt fig03_06.cpp format.cc -o
```

3. Type the following command to execute the program:

```
./fig03_06
```

## Compiling and Running the Example in Apple Xcode

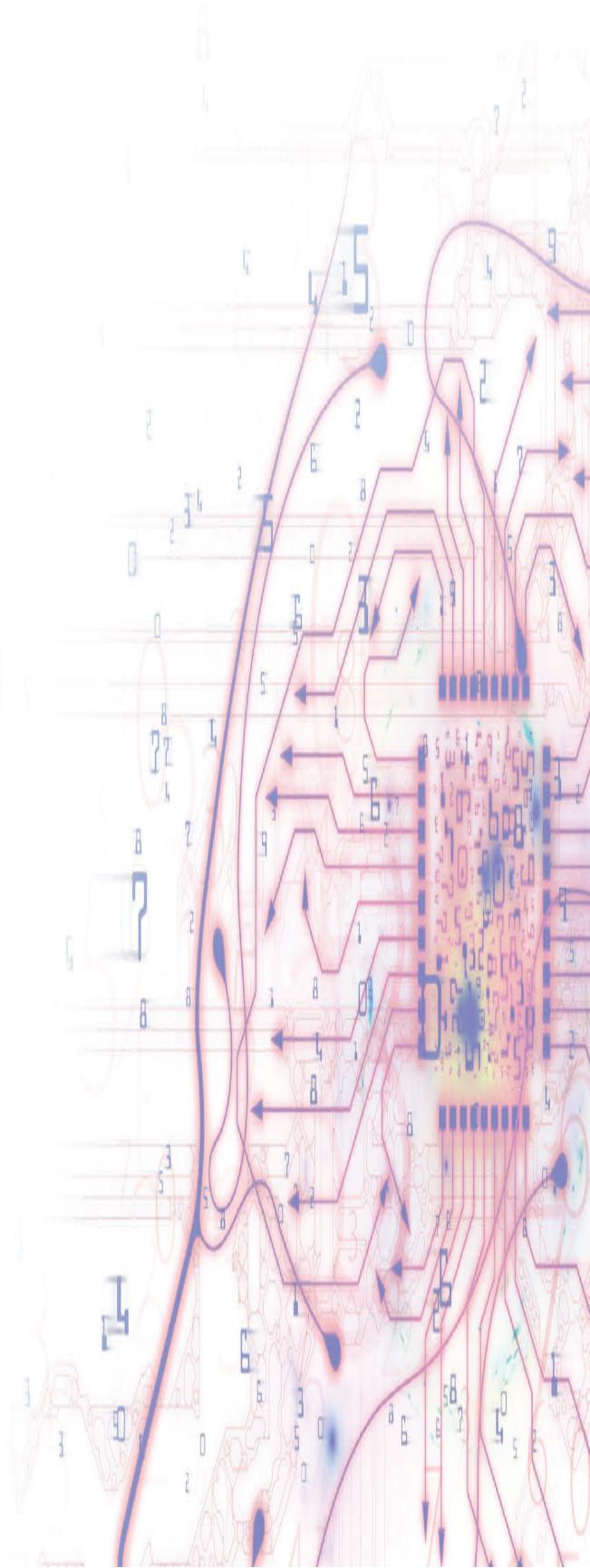
The steps for this example are the same as those in Section 3.12 with the following change:

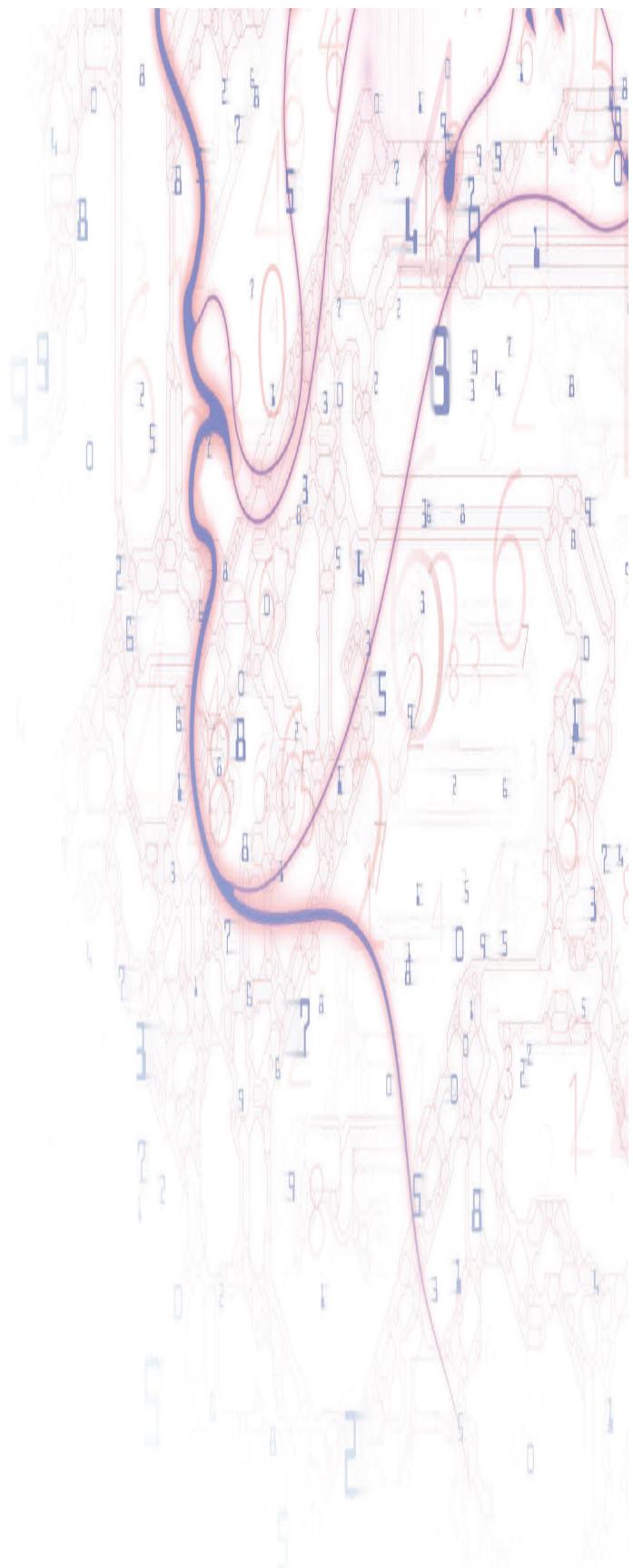
- In Step 2, drag the files `fig03_06.cpp`, `format.cc` and the folder `fmt` from to the `fig03_06` folder onto your project's source code folder in Xcode.

## 3.14 WRAP-UP

Only three types of control statements—sequence, selection and iteration—are needed to develop any algorithm. We demonstrated the `if` single-selection statement, the `if...else` double-selection statement and the `while` iteration statement. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled iteration, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced C++’s compound assignment operators and its increment and decrement operators. We discussed why C++’s fundamental types are not portable, then used objects of the open-source class `BigNumber` to perform integer arithmetic with values outside the range supported by our systems. Finally, we introduced C++20’s new text formatting in the context of the open-source `{fmt}` library. In [Chapter 4](#), we continue our discussion of control statements, introducing the `for`, `do...while` and `switch` statements, and we introduce the logical operators for creating compound conditions.

## 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.
- Use C++20's `[[likely]]` and `[[unlikely]]` attributes to help the compiler optimize selection statements by knowing which paths of execution are likely or unlikely to execute.
- 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](#)

[4.2 Essentials of Counter-Controlled Iteration](#)

[4.3 `for` Iteration Statement](#)

[4.4 Examples Using the `for` Statement](#)

[4.5 Application: Summing Even Integers](#)

[4.6 Application: Compound-Interest Calculations](#)

- [\*\*4.7\*\* do...while Iteration Statement](#)
  - [\*\*4.8\*\* switch Multiple-Selection Statement](#)
  - [\*\*4.9\*\* C++17: Selection Statements with Initializers](#)
  - [\*\*4.10\*\* break and continue Statements](#)
  - [\*\*4.11\*\* Logical Operators
    - \[4.11.1 Logical AND \\(&&\\) Operator\]\(#\)
    - \[4.11.2 Logical OR \\(||\\) Operator\]\(#\)
    - \[4.11.3 Short-Circuit Evaluation\]\(#\)
    - \[4.11.4 Logical Negation \\(!\\) Operator\]\(#\)
    - \[4.11.5 Logical Operators Example\]\(#\)](#)
  - [\*\*4.12\*\* Confusing the Equality \(==\) and Assignment \(=\) Operators](#)
  - [\*\*4.13\*\* C++20 Feature Mock-Up: \[\[likely\]\] and \[\[unlikely\]\] Attributes](#)
  - [\*\*4.14\*\* Objects Natural Case Study: Using the `miniz-cpp` Library to Write and Read ZIP files](#)
  - [\*\*4.15\*\* C++20 Feature Mock-Up:](#)
  - [\*\*4.16\*\* Wrap-Up](#)
- 

## 4.1 INTRODUCTION

**17 20** 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 discuss the logical operators, which enable you to combine simple conditions to form compound conditions. We show C++20’s attributes `[[likely]]` and `[[unlikely]]`, which can help the compiler optimize

selection statements by knowing which paths of execution are likely or unlikely to execute. In our objects-natural case study, we continue using objects of pre-existing classes with the miniz-cpp open-source library for creating and reading compressed ZIP archive files. Finally, we introduce more of C++20's powerful and expressive text-formatting features.

### **“Rough-Cut” E-Book for O'Reilly Online Learning Subscribers**

You are viewing an early-access “rough cut” of *C++20 for Programmers*. We prepared this content carefully, but it has not yet been reviewed or copy edited and is subject to change. As we complete each chapter, we'll post it here. Please send any corrections, comments, questions and suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I'll respond promptly. Check here frequently for updates.

### **“Sneak Peek” Videos for O'Reilly Online Learning Subscribers**

As an O'Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

<https://learning.oreilly.com/videos/c-20-fundame>



Co-author Paul Deitel immediately records each video lesson as we complete each rough-cut e-book chapter. Lessons go live on O'Reilly Online Learning a few days later. Again, check here frequently for updates.

## **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 the application of Fig. 4.1, which uses a loop to display the numbers from 1 through 10.

```
1 // fig04_01.cpp
2 // Counter-controlled iteration with the while iteration statement.
3 #include <iostream>
4 using namespace std;
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 << endl;
15 }
```

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

**Fig. 4.1** Counter-controlled iteration with the `while` iteration statement.

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. In C++, it's more precise to call a variable declaration that also reserves memory a **definition**. Because definitions are declarations, too, 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 program performs the `while`'s body even when the control variable is `10`. 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 tests for termination. For that reason, always control counting loops with integer values.

### 4.3 FOR ITERATION STATEMENT

C++ also provides the **for iteration statement**, which 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.

---

```
1 // fig04_02.cpp
2 // Counter-controlled iteration with the for iteration statement.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     // for statement header includes initialization,
8     // loop-continuation condition and increment
9     for (int counter[1]; counter <= 10; ++counter) {
10         cout << counter << " ";
11     }
12
13     cout << endl;
14 }
```

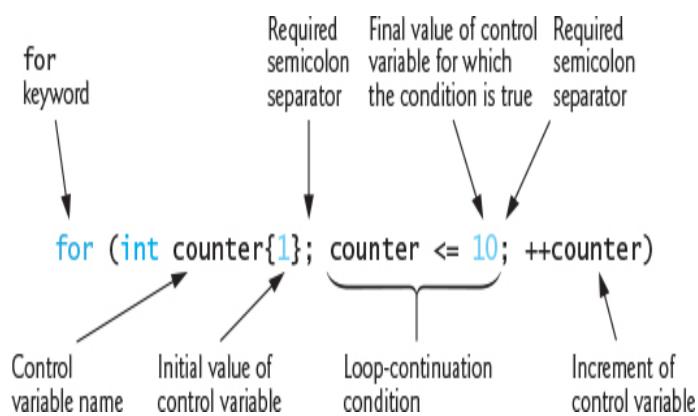
```
1 2 3 4 5 6 7 8 9 10
```

**Fig. 4.2** Counter-controlled iteration with the `for` iteration statement.

When the `for` statement (lines 9–11) begins executing, the control variable `counter` is declared and initialized to 1. Next, the program checks the loop-continuation condition, `counter <= 10`, which is between the two required semicolons. Because `counter`'s initial value is 1, the condition is true. So, the body statement (line 10) displays `counter`'s value (1). After executing the loop's body, the program increments `counter` in the expression `++counter`, which appears to the right of the second semicolon. 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, so the condition is still true, so the program executes the body statement 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 executing at the first statement after the `for` (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

- the *initialization* expression names the loop’s control variable and provides its initial value,
- the *loopContinuationCondition* determines whether the loop should continue executing and
- the *increment* modifies the control variable’s value so that the loop-continuation condition eventually becomes false.

The two semicolons in the `for` header are required. 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 in the `for` header 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 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. We discuss scope in detail in [Chapter 5](#).

### 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, thus 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 standalone 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. Many programmers prefer `counter++` because it's concise and because a `for` loop evaluates its increment expression after its body executes, so the postfix increment form seems more natural. 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 14](#)'s operator overloading discussion, you'll see that preincrement can have a performance advantage.

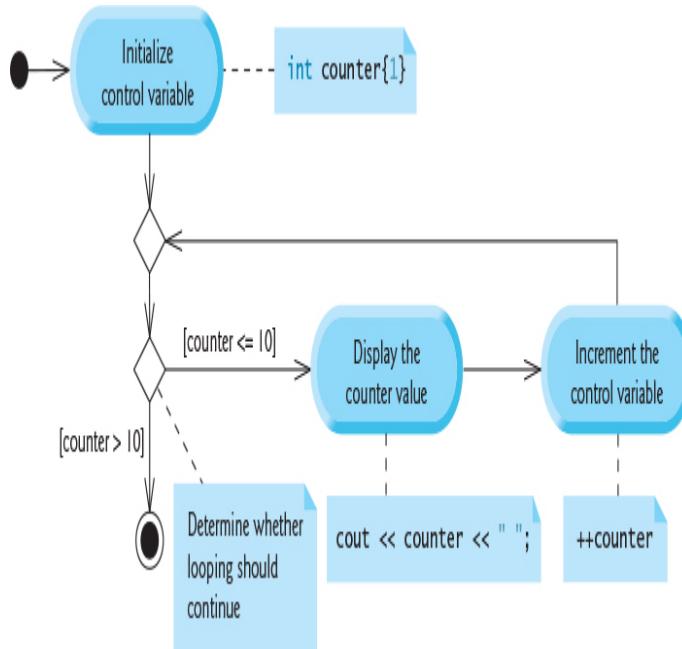
### Using a `for` Statement's Control Variable in the Statement's Body

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The control variable is commonly used to control iteration without being mentioned in the body of the `for`.

Although the value of the control variable can be changed in a `for` loop's body, avoid doing so, because this practice can lead to subtle errors. If a program must modify the control variable's value in the loop's body, use `while` rather than `for`.

### UML Activity Diagram of the `for` Statement

Below is the UML activity diagram of the `for` statement in [Fig. 4.2](#):



The diagram makes it clear that initialization occurs only once—before testing the loop-continuation condition the first time. Incrementing occurs each time through the loop after the body statement executes.

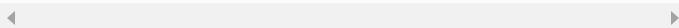
#### 4.4 EXAMPLES USING THE `FOR` STATEMENT

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.

---

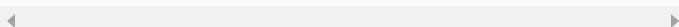
```
for (int i{1}; i <= 100; ++i)
```



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

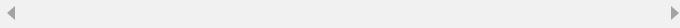
---

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



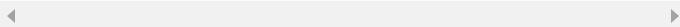
- c) Vary the control variable from 7 to 77 in increments of 7.

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



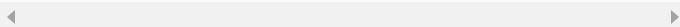
- d) Vary the control variable from 20 down to 2 in decrements of 2.

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



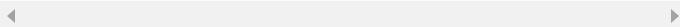
- e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

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



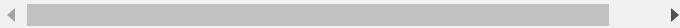
- f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

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

```
Sum is 110
```

**Fig. 4.3** Summing integers with the `for` statement. (Part 1 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

```
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. For example, assuming `x` is an `int` and `y` is a `double`, the value of the comma-separated list of expressions

```
x = 5, y = 6.4;
```

is `6.4` and the type is `double`.

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

*a* is the amount on deposit at the end of the *n*th year.

The solution to this problem (Fig. 4.4) involves a loop that performs the indicated 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. In [Chapter 10](#), we'll develop a new **Dollar-Amount** class that uses large integers to precisely represent monetary amounts. As you'll see, the class performs monetary calculations using only integer arithmetic.

---

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 << endl;
16    cout << "    Interest rate: " << rate << endl;
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount on deposit" << endl;
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 << endl;
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`. (Part 1 of 2.)

Lines 12–13 declare `double` variables `principal` and `rate`, and initialize `principal` to `1000.00` and `rate` to `0.05`. C++ treats floating-point literals like `1000.00` and `0.05` as type `double`. Similarly, C++ treats whole-number literals like `7` and `-22` as type `int`.<sup>1</sup> Lines 15–16 display the initial principal and the interest rate.

1. 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 is fewer than four character positions, the value is right-aligned in the field by default. If the value to be output is 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 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—such settings are called **sticky settings**, 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 15 discusses `cin`’s and `cout`’s formatting capabilities in detail. We continue discussing C++20’s powerful new text-formatting capabilities in Section 4.15.

### 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  $y$ th 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`.



The body of the `for` statement contains the calculation `1.0 + rate` as `pow`’s first argument. This calculation produces the same result each time through the loop, so repeating it in every iteration of the loop is wasteful. In loops, avoid calculations for which the result never changes. Instead, place such calculations before the loop. To improve program performance, many of today’s optimizing compilers place such calculations before loops in the compiled code.

### Floating-Point Number Precision and Memory Requirements

Variables of type `float` represent **single-precision floating-point numbers**. Most of today’s systems store these in four bytes of memory with approximately seven significant digits. Variables of type `double` represent **double-precision floating-point numbers**. Most of today’s systems store these in eight bytes of memory with approximately 15 significant

digits—approximately double the precision of `float` variables. Most programmers represent floating-point numbers with type `double`. C++ treats floating-point numbers like 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**.

Though most systems store `floats` in four bytes and `doubles` in eight bytes, the C++ standard indicates that type `double` provides 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—when we divide 10 by 3, the result is 3.333333..., with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation. As you can see, `double` suffers 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.

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 98.5999473210643. Calling this number 98.6 is fine for most applications involving body temperatures. Generally, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more precisely. We use `double` throughout the book.

## A Warning about Displaying Rounded Values

We declared variables `amount`, `principal` and `rate` to be of type `double` in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using floating-point numbers 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

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

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 Representational Error in Floating Point

Even simple dollar amounts, such as those you might see on a grocery or restaurant bill, can have representational errors when they're stored as `doubles`. To see this, we created a simple program with the declaration

```
double d = 123.02;
```

then displayed variable `d`'s value with many 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`, 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

The **do...while iteration statement** is similar to the **while** statement. In a **while** statement, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body. If the condition is false, the body never executes. The **do...while** statement tests the loop-continuation condition after executing the loop's body; therefore, the body always executes at least once. When a **do...while** statement terminates, execution continues with the next statement in sequence. Figure 4.5 uses a **do...while** to output the numbers 1–10.

```
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 << endl;
15 }
```

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

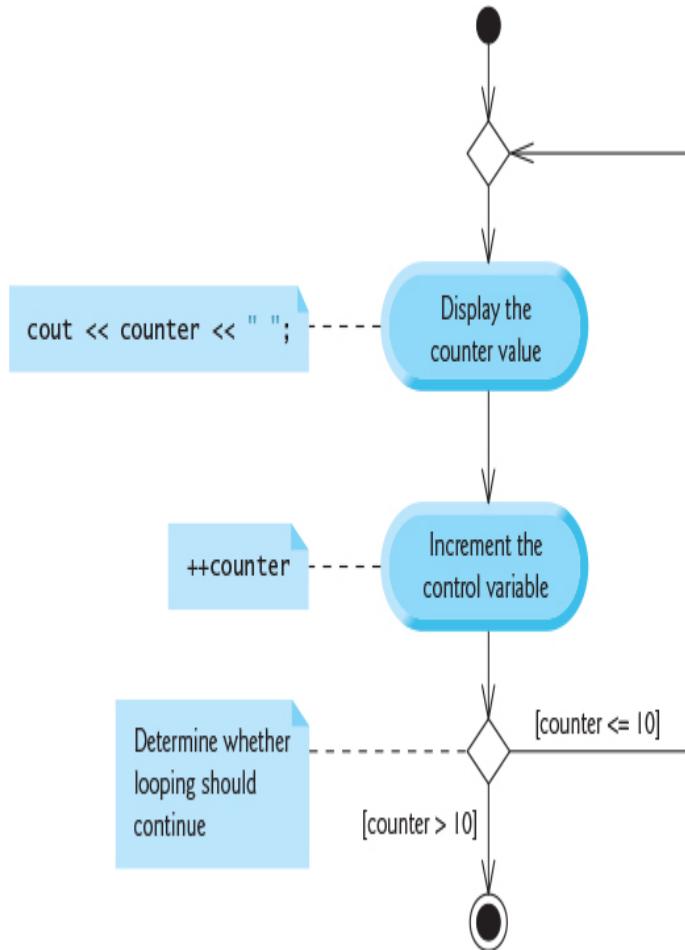
**Fig. 4.5** **do...while** iteration statement.

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.

### UML Activity Diagram for the `do...while` Iteration Statement

The UML activity diagram for the `do...while` statement in Fig. 4.5 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` (i.e., 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.

---

```
1 // fig04_06.cpp
2 // Using a switch statement to count letter grades.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main() {
8     int total{0}; // sum of grades
```

```
9  int gradeCounter{0}; // number of grades entered
10 int aCount{0}; // count of A grades
11 int bCount{0}; // count of B grades
12 int cCount{0}; // count of C grades
13 int dCount{0}; // count of D grades
14 int fCount{0}; // count of F grades
15
16 cout << "Enter the integer grades in the range 0-100.\n"
17   << "Type the end-of-file indicator to terminate input:\n"
18   << "  On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
19   << "  On Windows type <Ctrl> z then press Enter\n";
20
21 int grade;
22
23 // loop until user enters the end-of-file indicator
24 while (cin >> grade) {
25     total += grade; // add grade to total
26     ++gradeCounter; // increment number of grades
27
28     // increment appropriate letter-grade counter
29     switch (grade / 10) {
30         case 9: // grade was between 90
31         case 10: // and 100, inclusive
32             ++aCount;
33             break; // exits switch
34
35         case 8: // grade was between 80 and 89
36             ++bCount;
37             break; // exits switch
38
39         case 7: // grade was between 70 and 79
40             ++cCount;
41             break; // exits switch
42
43         case 6: // grade was between 60 and 69
44             ++dCount;
45             break; // exits switch
46
47         default: // grade was less than 60
48             ++fCount;
49             break; // optional; exits switch anyway
50     } // end switch
```

```
51 } // end while
52
53 // set floating-point number format
54 cout << fixed << setprecision(2);
55
56 // display grade report
57 cout << "\nGrade Report:\n";
58
59 // if user entered at least one grade...
60 if (gradeCounter != 0) {
61     // calculate average of all grades entered
62     double average = static_cast<double>(total) / gradeCounter;
```

```

63
64 // output summary of results
65 cout << "Total of the " << gradeCounter << " grades entered is "
66     << total << "\nClass average is " << average
67     << "\nNumber of students who received each grade:"
68     << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69     << "\nD: " << dCount << "\nF: " << fCount << endl;
70 }
71 else { // no grades were entered, so output appropriate message
72     cout << "No grades were entered" << endl;
73 }
74 }
```

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

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 1 of 3.)

The `main` function (Fig. 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, respectively. Lines 10–14 declare and initialize to 0 counter variables for each grade category. The `main` function has two key parts. 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 9, File Processing, 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`

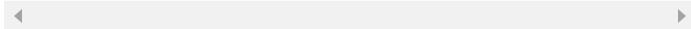
On some systems, you must press *Enter* after typing the end-of-file key sequence. Also, Windows typically displays the characters `^Z` on the screen when you type the end-of-file indicator, as shown in the output of Fig. 4.6.

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 evaluates to true, line 25 adds `grade` to `total` and line 26 increments `gradeCounter`. These variables are used to compute the average of the grades. 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.

**11** 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`, `char16_t`, `char32_t`, `wchar_t`, `int`, `long` or `long long`. The expression can also use the C++11 signed or unsigned integral types, such as `int64_t` and `uint64_t`—see the `<cstdint>` header for a complete list of these type names.

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) exists 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 `cases` 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`.

### case without a break Statement

Without `break` statements, each time a match occurs in the `switch`, the statements for that case and subsequent cases execute until a `break` statement or the end of the `switch` is encountered. This is often referred to as “falling through” to the statements in subsequent `cases`.<sup>2</sup>

2. 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.

### C++17: [[fallthrough]] Attribute

**17** Forgetting a `break` statement when one is needed is a logic error. To call your attention to this possible problem, many compilers issuing a warning when a `case` does not

contain a `break` statement. For instances in which “falling through” is the desired behavior, C++17 introduced the **`[[fallthrough]]` attribute**. This enables you to tell the compiler when “falling through” is correct so that warning will not be generated.

In Fig. 4.6, for `case 9:` (line 30), we want the `switch` to fall through (without a compiler warning) and execute the statements for `case 10:`—this allows both cases to execute the same statements. We can indicate the desired behavior by writing line 30 as:

```
case 9: [[fallthrough]];
```

### The default Case

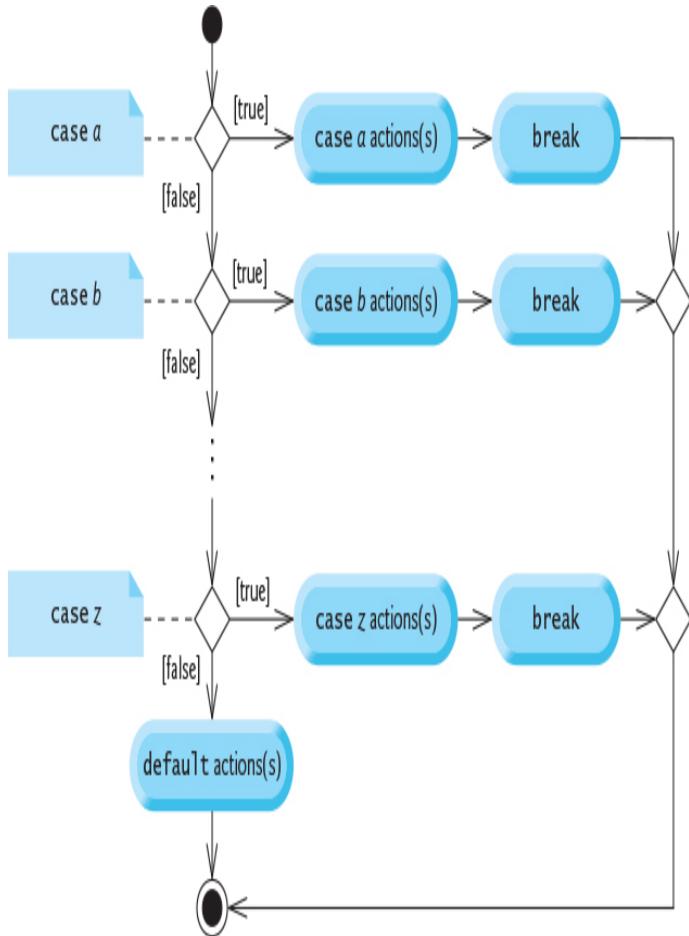
If no match occurs between the controlling expression’s value and any of the `case` labels, the `default` case (lines 47–49) executes. We use the `default` case in this example to process all controlling-expression values that are less than 6—that is, all failing grades. If no match occurs and the `switch` 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` statement after processing the `case`. 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 combination of integer

constants that evaluates to a constant integer value. An integer constant is simply an integer value. You also can use `enum` constants (introduced in Section 5.8) and **character constants**—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.)

The expression in each `case` also can be a **constant variable**—a variable containing a value that does not change for the entire program. Such a variable is declared with keyword `const` (discussed in [Chapter 5](#)).

In [Chapter 13](#), Object-Oriented Programming: 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.

## 17 4.9 C++17: SELECTION STATEMENTS WITH INITIALIZERS

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 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.

---

```
1 // fig04_07.cpp
2 // C++17 if statements with initializers.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     if (int value{7}; value == 7) {
8         cout << "value is " << value << endl;
9     }
10    else {
11        cout << "value is not 7; it is " << value << endl;
12    }
13
14    if (int value{13}; value == 9) {
15        cout << "value is " << value << endl;
16    }
17    else {
18        cout << "value is not 9; it is " << value << endl;
19    }
20 }
```

```
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 statements `break` and `continue` 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

The `break` statement, when executed 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 a `for` early.

---

```
1 // fig04_08.cpp
2 // break statement exiting a for statement.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int count; // control variable also used after loop
8
9     for (count = 1; count <= 10; ++count) { // loop 10 times
10        if (count == 5) {
11            break; // terminates for loop if count is 5
12        }
13
14        cout << count << " ";
15    }
16
17    cout << "\nBroke out of loop at count = " << count << endl;
18 }
```

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

**Fig. 4.8** break statement exiting a for statement. (Part 1 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.

### **continue Statement**

The **continue** statement, when executed 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.

---

```
1 // fig04_09.cpp
2 // continue statement terminating an iteration of a for statement.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     for (int count{1}; count <= 10; ++count) { // loop 10 times
8         if (count == 5) {
9             continue; // skip remaining code in loop body if count is 5
10        }
11
12         cout << count << " ";
13    }
14
15    cout << "\nUsed continue to skip printing 5" << endl;
16 }
```

```
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 do not use `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 `if`, `if...else`, `while`, `do...while` and `for` statements each require a condition to 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 in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`, and each expression tests only one condition. To test multiple conditions in the process of making a decision, we performed these tests in separate statements or in nested `if` or `if...else` statements. Sometimes control statements require more complex conditions to determine a program's flow of control.

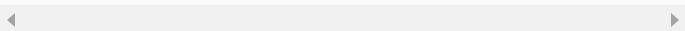
C++'s **logical operators** enable you to combine simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical negation).

### 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;  
}
```



This `if` statement contains two simple conditions. The condition `gender == FEMALE` compares variable `gender` to the constant `FEMALE` to determine 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 table summarizes the `&&` operator, showing all four possible combinations of the `bool` values `false` and `true` values for expression1 and expression2:

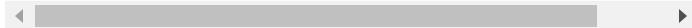
expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Such tables are called **truth tables**. C++ evaluates to zero (false) or nonzero (true) all expressions that include relational operators, equality operators or logical operators.

### 4.11.2 Logical OR (`||`) Operator

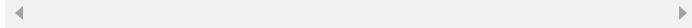
Now suppose we wish to ensure that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the `||` (logical OR) operator, as in the following program segment:

```
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 (`||`):

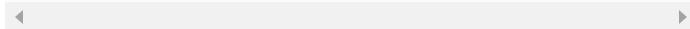
expression1	expression2	expression1    expression2
false	false	false
false	true	true
true	false	true
true	true	true

Operator `&&` has higher precedence than operator `||`. 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 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 as follows:

```
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: Using the Logical Operators to Produce Their 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. The sticky **stream manipulator `boolalpha`** (line 8) specifies that each `bool` expression's value should be displayed as the word “true” or the word “false.” Lines 8–12, 15–19 and 22–24 produce the truth tables for `&&`, `||` and `!`, respectively.

---

```
1 // fig04_10.cpp  
2 // Logical operators.  
3 #include <iostream>  
4 using namespace std;  
5  
6 int main() {  
7     // create truth table for && (logical AND) operator
```

```

8   cout << boolalpha << "Logical AND (&&)"
9   << "\nfalse && false: " << (false && false)
10  << "\nfalse && true: " << (false && true)
11  << "\ntrue && false: " << (true && false)
12  << "\ntrue && true: " << (true && true) << "\n\n";
13
14 // create truth table for || (logical OR) operator
15 cout << "Logical OR (||)"
16 << "\nfalse || false: " << (false || false)
17 << "\nfalse || true: " << (false || true)
18 << "\ntrue || false: " << (true || false)
19 << "\ntrue || true: " << (true || true) << "\n\n";
20
21 // create truth table for ! (logical negation) operator
22 cout << "Logical negation (!)"
23 << "\n!false: " << (!false)
24 << "\n!true: " << (!true) << endl;
25 }

```

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. (Part 1 of 2.)

## 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	Type
:: ()	left to right <i>[See caution in Chapter 2 regarding grouping parentheses.]</i>	primary
++ -- static_cast<type>()	left to right	unary (postfix)
++ -- + - !	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

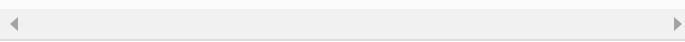
## 4.12 CONFUSING THE EQUALITY (==) AND ASSIGNMENT (=) OPERATORS

There's one 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 syntax errors. Statements with these errors tend to compile correctly and run to completion, often generating incorrect results through runtime logic errors. Some compilers issue a warning when = is used in a context where == is expected.

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 assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

---

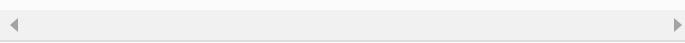
```
if (payCode == 4) { // good
    cout << "You get a bonus!" << endl;
}
```



but we accidentally write

---

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



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!

## Ivalues and rvalues

You can prevent the preceding 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”) because they can be used on only an assignment operator’s right side. *Lvalues* also can be used as *rvalues* on the right side of an assignment, 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), 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.

## 20 4.13 C++20 FEATURE MOCK-UP: [[LIKELY]] AND [[UNLIKELY]] ATTRIBUTES

 **PERF** Today's compilers use sophisticated optimization techniques<sup>3</sup> to tune your code's performance. C++20 introduces the attributes `[[likely]]` and `[[unlikely]]` that enable you to provide additional hints to help compilers

optimize `if`, `if...else` and `switch` statement code for better performance.<sup>4</sup> These attributes indicate paths of execution that are likely or unlikely to be taken. Many of today's compilers already provide mechanisms like this, so `[[likely]]` and `[[unlikely]]` standardize these features across compilers.<sup>5</sup>

3. "Optimizing Compiler." Wikipedia. Wikimedia Foundation, April 7, 2020. [https://en.wikipedia.org/wiki/Optimizing\\_compiler#Specific\\_techniques](https://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques).

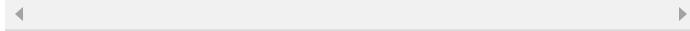
4. Note to reviewers: At the time of this writing, these attributes were not implemented by our preferred compilers. We searched for insights as to why and how you'd use this feature to produce better optimized code. Even though the standard is about to be accepted in May, there is little information at this point other than the proposal document "Attributes for Likely and Unlikely Statements (Revision 2)" (<https://wg21.link/p0479r2>). Section 3, Motivation and Scope, suggests who should use these features and what they should be used for.

5. Sutter, Herb. "Trip Report: Winter ISO C Standards Meeting (Jacksonville)." Sutter's Mill, April 3, 2018. <https://herbsutter.com/2018/04/>. Herb Sutter is the Convener of the ISO C++ committee and a Software Architect at Microsoft.

To use these attributes, place `[[likely]]` or `[[unlikely]]` before the body of an `if` or `else`, as in

---

```
if (condition) [[likely]] {
    // statements
}
else {
    // statements
}
```



or before a `case` label in a `switch`, as in

---

```
switch (controllingExpression) {
    case 7:
        // statements
        break;
    [[likely]] case 11:
        // statements
        break;
    default:
        // statements
        break;
}
```





**PERF** There are subtle issues when using these attributes.

Using too many `[[likely]]` and `[[unlikely]]` attributes in your code could actually reduce performance.<sup>6</sup>

The document that proposed adding these to the language says for each, “This attribute is intended for specialized optimizations which are implementation specific. General usage of this attribute is discouraged.”<sup>7</sup> For a discussion of other subtleties, see the proposal document at:

<https://wg21.link/p0479r2>

◀ ▶

6. Section 9.12.6, “Working Draft, Standard for Programming Language C.”

ISO/IEC, April 3, 2020.

<https://github.com/cplusplus/draft/releases/download/n4861/n4861.pdf>.

7. “Attributes for Likely and Unlikely Statements (Revision 2).”

<https://wg21.link/p0479r2>. Section VIII, Technical Specifications.

If you’re working on systems with strict performance requirements you may want to investigate these attributes further.

## 4.14 OBJECTS NATURAL CASE STUDY: USING THE `MINIZ-CPP` LIBRARY TO WRITE AND READ ZIP FILES<sup>8</sup>

8. This example does not compile in GNU C++.



**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 audio and video. For example, when you watch streaming video online, the video is often compressed 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**<sup>9</sup> is a lossless compression<sup>10</sup> 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.

<sup>9</sup>. “Zip (File Format).” Wikipedia. Wikimedia Foundation, April 23, 2020.  
[https://en.wikipedia.org/wiki/Zip\\_\(file\\_format\)](https://en.wikipedia.org/wiki/Zip_(file_format)).

<sup>10</sup>. “Data Compression.” Wikipedia. Wikimedia Foundation, April 16, 2020.  
[https://en.wikipedia.org/wiki/Data\\_compression#Lossless](https://en.wikipedia.org/wiki/Data_compression#Lossless).

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.<sup>11</sup> Figure 4.11 continues our objects natural presentation by using objects of the open-source `miniz-cpp`<sup>12,13</sup> 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` that you can simply include in your project (line 5). We provide the library in the `examples` folder’s `libraries/minizcpp` subfolder. Header files are discussed in depth in Chapter 10.

11. "List of Archive Formats." Wikipedia. Wikimedia Foundation, March 19, 2020.  
[https://en.wikipedia.org/wiki/List\\_of\\_archive\\_formats](https://en.wikipedia.org/wiki/List_of_archive_formats)
12. <https://github.com/tfussell/miniz-cpp>.
13. The miniz-cpp library provides nearly identical capabilities 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.
- 

```
1 // fig04_11.cpp
2 // Using the miniz-cpp header-only library to write and read a ZIP file.
3 #include <iostream>
4 #include <string>
5 #include "zip_file.hpp"
6 using namespace std;
7
```

---

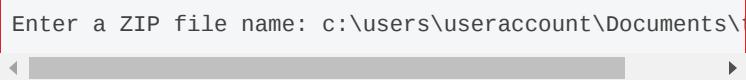
**Fig. 4.11** Using the miniz-cpp header-only library to write and read a ZIP file.

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

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

---

```
8 int main() {
9     cout << "Enter a ZIP file name: ";
10    string zipFileName;
11    getline(cin, zipFileName); // inputs a line of
text
12
```



A screenshot of a terminal window. A red box highlights the text "Enter a ZIP file name: c:\users\useraccount\Documents\\" which has been typed by the user. Below the terminal window, there is a horizontal scrollbar with arrows at both ends.

Here we use `getline` to read from the user a 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 into Individual Files in the ZIP File

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

---

```
13    // strings literals separated only by whitespace  
14    // are combined  
15    string content{  
16        "This chapter introduces all but one of the  
17        remaining control "  
18        "statements--the for, do...while, switch,  
19        break and continue "  
20        "statements. We explore the essentials of  
21        counter-controlled "  
22        "iteration. We use compound-interest  
23        calculations to begin "  
24        "investigating the issues of processing  
25        monetary amounts. First, "  
26        "we discuss the representational errors  
27        associated with "  
28        "floating-point types. We use a switch  
29        statement to count the "  
30        "number of A, B, C, D and F grade equivalents  
31        in a set of "  
32        "numeric grades. We show C++17's enhancements  
33        that allow you to "  
34        "initialize one or more variables of the same  
35        type in the "  
36        "headers of if and switch statements."};  
37
```

---

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(): " <<  
29    content.length();  
30
```

---

```
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
```

**Fig. 4.11** Using the `miniz-cpp` header-only library to write and read a ZIP file. (Part 6 of 10)

### ZIP 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:

PKIII ॥ ॥ t-pNÉ ol y intro.txt]’KnÜö†÷‡±  
tōu „” (D5-#¢#‡ä’OL’çEçCéf Èoëéüøwd..ñ”\$A¹JY[ ॥ -V;d,²AIIß;ra|P|1  
Z±ÖN¹ëÄùv/d,,üÍd<ch|%q  
q,,EÜy æÜrnöévt?loÜyHE”fHØp3&ut (-, éM’lIIX¹ä~Ü”li?loØWI+ll;¥%\*p!,tákþú/öp|8EUlu!År«lv³]Qí  
€»Ómá!l<sup>o</sup>q, l-,-l@) “9+fb, IIIHKOµlf+X qØréÁé(&leý ýØž~ä#lxÙm; dÈm\_-Søc,, ý#<ð<sup>o</sup>gù lAðóúxØšE’  
ý{dAm-HåáÜ;yÜ  
žÜž)IS 1†³#äööm~/ÍxVi%ö+X, öDl...qigf, åNÝ’A]p\$äiÝöö•~5” ööPKö ॥ ॥ t-pNÉ ol y  
intro.txtPKII ॥ ॥ 7 -॥

## 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	04/23/2020	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 filen in the archive. The object `info` contains information about the archive's `intro.txt` file, including the file's name (`info.filename`), its uncom-pressed 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 only 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 a compressed file from the ZIP archive to restore the original. 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`:

```
50 // original file contents
51 string extractedContent{input.read(info)};
52
```

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

```
53 cout << "\n\nOriginal contents of intro.txt:\n"
<<
54     extractedContent << endl;
55 }
```

Original contents of intro.txt:  
This chapter introduces all but one of the remaining control statements: `for`, `do...while`, `switch`, `break` and `continue`. These are the essentials of counter-controlled iteration. We use compound assignment operators to make calculations to begin investigating the issues of precision in floating-point numbers. First, we discuss the representational errors in floating-point types. We use a `switch` statement to convert letter grades C, D and F grade equivalents in a set of numeric grades. Finally, we introduce enhancements that allow you to initialize one or more variables of a type in the headers of `if` and `switch` statements.

## 20 4.15 C++20 FEATURE MOCK-UP: TEXT FORMATTING WITH FIELD WIDTHS AND PRECISIONS

In Section 3.13, we 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.<sup>14</sup> 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.

14. Some of our C++20 Feature Mock-Up sections present code that does not compile or run. Once the compilers implement those features, we'll retest the code, update our digital products and post updates for our print products at <https://deitel.com/c-plus-plus-20-for-programmers>. The code in this example runs, but uses the `{fmt}` open-source library to demonstrate features that C++20 compilers will support eventually.

---

```
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
13    cout << format("Initial principal: {:.2f}\n", principal)
14      << format("    Interest rate: {:.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

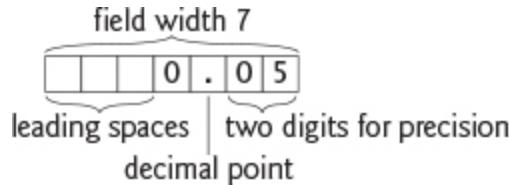
Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

**Fig. 4.12** Compound-interest example with C++20 string formatting.

### 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 field width of 7 position with two digits of precision (`.2`)—that is, two positions 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"
```

"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) right-aligned (`>`) in a field of width

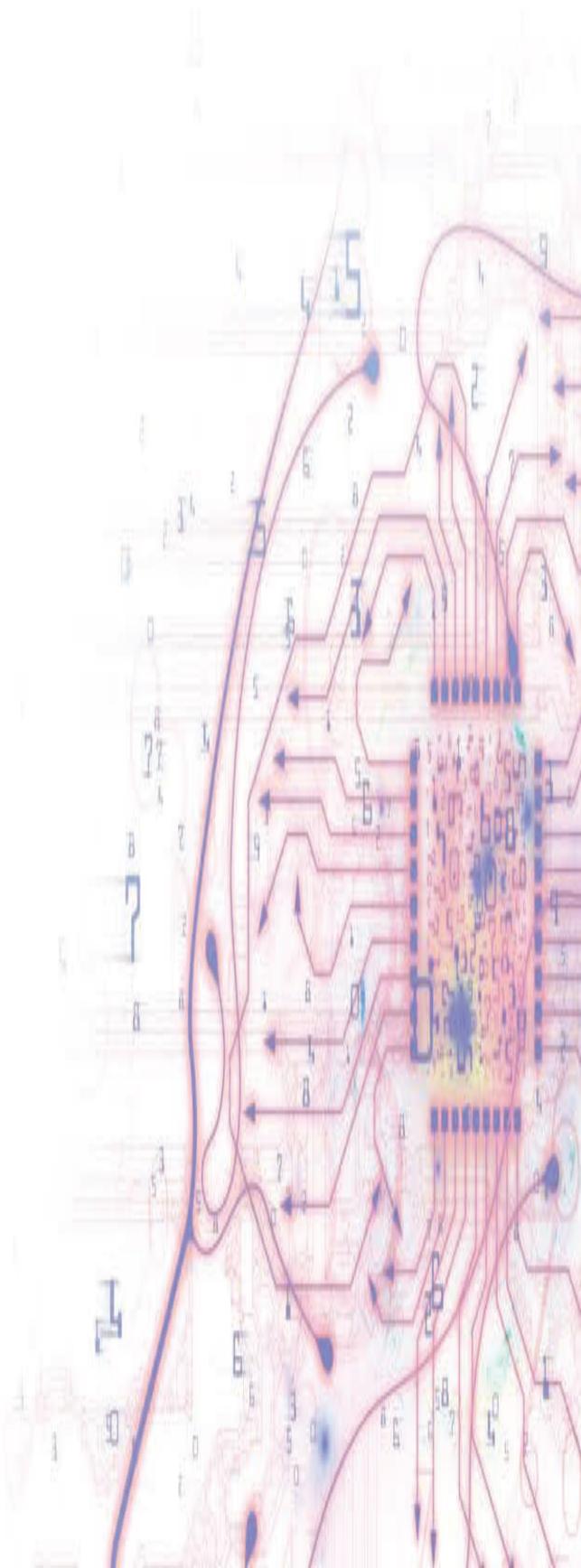
4. This right-aligns all the year values under the "Year" column.

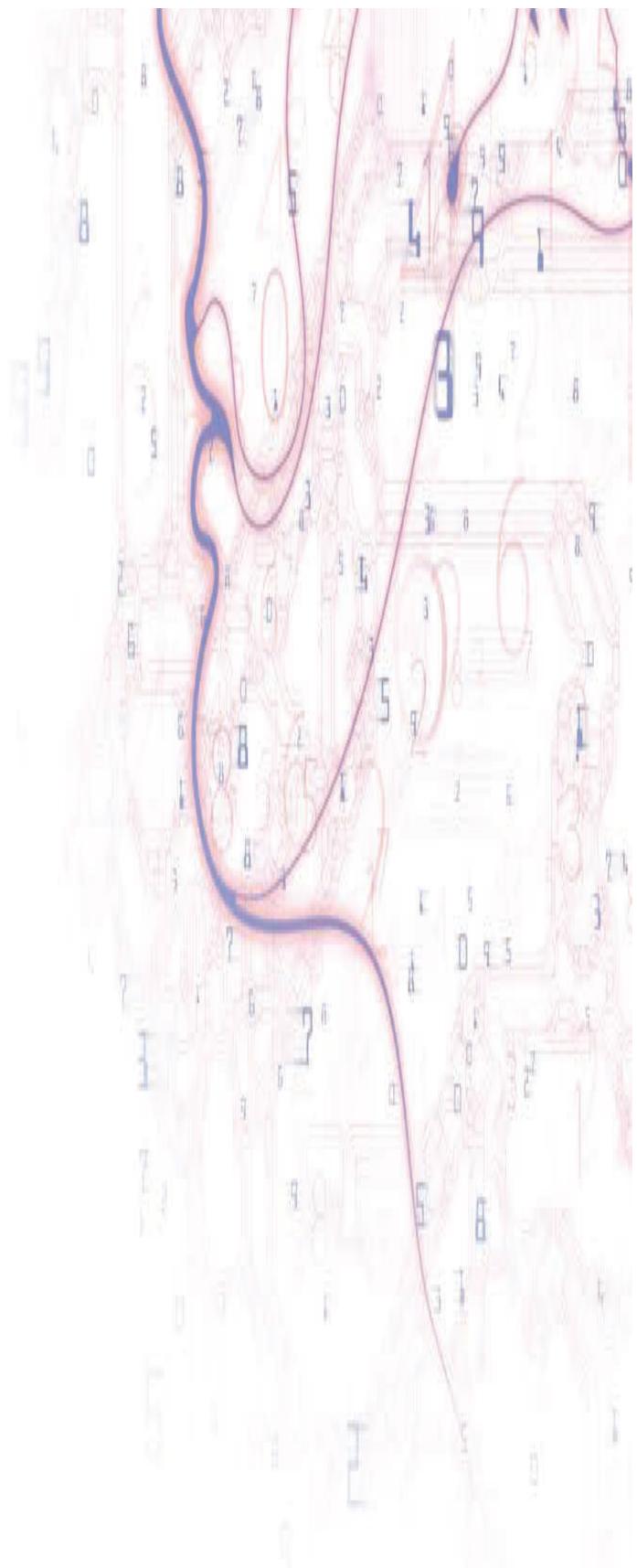
The placeholder `{ :>20.2f}` formats `amount`'s value as a floating-point number (`f`) right-aligned (`>`) in a field width of `20` with a decimal point and two digits to the right of the decimal point (`.2`). Formatting the `amounts` this way *aligns their decimal points vertically*, as is typical with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

## 4.16 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 member 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. We showed C++20's attributes `[[likely]]` and `[[unlikely]]` for hinting to the compiler which paths of execution are likely or unlikely to execute in selection statements. In our 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.

## 5. Functions





## Objectives

In this chapter, you'll:

- Construct programs modularly from functions.
- Use common math library functions and learn about math functions and constants added in C++20, C++17 and C++11.
- Declare functions with function prototypes.
- View many key C++ Standard Library headers.
- Use random numbers to implement game-playing apps.
- Declare constants in scoped `enums` and use constants without their type names via C++20's `using enum` declarations.
- Make long numbers more readable with digit separators.
- Understand the scope of identifiers.
- Use inline functions, references and default arguments.
- Define overloaded functions that perform different tasks based on the number and types of their arguments.
- Define function templates that can generate families of overloaded functions.
- Write and use recursive functions.
- Use the C++17 and C++20 `[[nodiscard]]` attribute to indicate that a function's return value should not be ignored.
- Zajnropc vrq lnfylun-lhqtmh uyqmmhzg tupb j dvql psrpu iw dmwwqnndwjzqz.

---

## Outline

- 5.1** Introduction
- 5.2** Program Components in C++
- 5.3** Math Library Functions
- 5.4** Function Definitions and Function Prototypes
- 5.5** Order of Evaluation of a Function's Arguments
- 5.6** Function-Prototype and Argument-Coercion Notes
  - 5.6.1** Function Signatures and Function Prototypes
  - 5.6.2** Argument Coercion
  - 5.6.3** Argument-Promotion Rules and Implicit Conversions
- 5.7** C++ Standard Library Headers
- 5.8** Case Study: Random-Number Generation
  - 5.8.1** Rolling a Six-Sided Die
  - 5.8.2** Rolling a Six-Sided Die 60,000,000 Times
  - 5.8.3** Randomizing the Random-Number Generator with `srand`
  - 5.8.4** Seeding the Random-Number Generator with the Current Time
  - 5.8.5** Scaling and Shifting Random Numbers
- 5.9** Case Study: Game of Chance; Introducing Scoped `enums`
- 5.10** C++11's More Secure Nondeterministic Random Numbers
- 5.11** Scope Rules
- 5.12** Inline Functions
- 5.13** References and Reference Parameters
- 5.14** Default Arguments
- 5.15** Unary Scope Resolution Operator
- 5.16** Function Overloading
- 5.17** Function Templates
- 5.18** Recursion
- 5.19** Example Using Recursion: Fibonacci Series
- 5.20** Recursion vs. Iteration
- 5.21** C++17 and C++20: `[ [nodiscard] ]` Attribute
- 5.22** Lnfyln Lhqtmh Wjtz Qarcv: Qjwazkrplm xzz  
Xndmwqhlz

## **5.23** Wrap-Up

---

### **5.1 INTRODUCTION**

In this chapter, we introduce custom functions. We overview a portion of the C++ Standard Library’s math functions and introduce new functions and constants added in C++20, C++17 and C++11. We introduce function prototypes and discuss how the compiler uses them, if necessary, to convert the type of an argument in a function call to the type specified in a function’s parameter list. We also present an overview of the C++ Standard Library’s headers.

Next, we demonstrate simulation techniques with random number generation. We develop a version of a popular casino dice game that uses most of the C++ capabilities we’ve presented so far. In the game, we show how to declare constants in scoped `enums` and discuss C++20’s new `using enum` declarations for accessing scoped `enum` constants directly without their type name.

We then present C++’s scope rules, which determine where identifiers can be referenced in a program. We discuss features that help improve program performance—inline functions that can eliminate the overhead of a function call and reference parameters that can be used to pass large data items to functions efficiently.

Many of the applications you develop will have more than one function of the same name. This technique, called function overloading, is used to implement functions that perform similar tasks for arguments of different types or different numbers of arguments. We consider function templates—a mechanism for concisely defining a family of overloaded functions. We introduce recursive functions that call themselves, either directly, or indirectly through another function.

We present C++17’s `[ [nodiscard] ]` attribute for indicating that a function’s return value should not be ignored. This helps compilers warn you when the return value is not

used in your program. We also discuss C++20’s `[nodiscard]` enhancement that allows you to specify a reason why the return value should not be ignored. Cujuumt, ul znkfehdf jsy lagqynb-ovrbozi mljapvao thqt w wjtz qarcv aj wazkrvdqxbu.

### “Rough-Cut” E-Book for O’Reilly Online Learning Subscribers

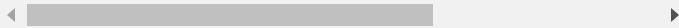
You are viewing an early-access “rough cut” of *C++20 for Programmers*. **We prepared this content carefully, but it has not yet been reviewed or copy edited and is subject to change.** As we complete each chapter, we’ll post it here. Please send any corrections, comments, questions and suggestions for improvement to [paul@deitel.com](mailto:paul@deitel.com) and I’ll respond promptly. Check here frequently for updates.

### “Sneak Peek” Videos for O’Reilly Online Learning Subscribers

As an O’Reilly Online Learning subscriber, you also have access to the “sneak peek” of our new *C++20 Fundamentals LiveLessons* videos at:

---

<https://learning.oreilly.com/videos/c-20-fundame>



Co-author Paul Deitel immediately records each video lesson as we complete each rough-cut e-book chapter. Lessons go live on O’Reilly Online Learning a few days later. Again, check here frequently for updates.

## 5.2 PROGRAM COMPONENTS IN C++

You typically write C++ programs by combining

- prepackaged functions and classes available in the C++ Standard Library,
- functions and classes available in a vast array of open-source and proprietary third-party libraries, and
- new functions and classes you and your colleagues write.

The C++ Standard Library provides a rich collection of functions and classes for math, string processing, regular expressions, input/output, file processing, dates, times, containers (collections of data), algorithms for manipulating containers, memory management, concurrent programming, asynchronous programming and many other operations.

Functions and classes allow you to separate a program's tasks into self-contained units. You've used a combination of C++ Standard Library features, open-source library features and the `main` function in every program so far. In this chapter, you'll begin defining custom functions, and starting in [Chapter 10](#), you'll define custom classes.

There are several motivations for using functions and classes to create program components:

- Software reuse. For example, in earlier programs, we did not have to define how to create and manipulate `strings` or how to read a line of text from the keyboard —C++ provides these capabilities via the `<string>` header's `string` class and `getline` function, respectively.
- Avoiding code repetition.
- Dividing programs into meaningful functions and classes makes programs easier to test, debug and maintain.

To promote reusability, every function should perform a single, well-defined task, and the function's name should express that task effectively. We'll say lots more about software reusability in our treatment of object-oriented programming. C++20 introduces another program component called modules, which we will discuss in later chapters.

### 5.3 MATH LIBRARY FUNCTIONS

In our objects-natural case study sections, you've created objects of interesting classes then called their member functions to perform useful tasks. Functions like `main` that are not member functions are called **global functions**.

The `<cmath>` header provides many global functions for common mathematical calculations. For example,

---

```
sqrt(900.0)
```

calculates the square root of `900.0` and returns the result, `30.0`. Function `sqrt` takes a `double` argument and returns a `double` result. There's no need to create any objects before calling function `sqrt`. All functions in the `<cmath>` header are global functions in the `std` namespace. Each is called simply by specifying the function name followed by parentheses containing the arguments.

Function arguments may be constants, variables or more complex expressions. Some popular math library functions are summarized in the following table, where the variables `x` and `y` are of type `double`.

Function	Description	Example
<code>ceil(x)</code>	rounds $x$ to the smallest integer not less than $x$	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>cos(x)</code>	trigonometric cosine of $x$ ( $x$ in radians)	<code>cos(0.0)</code> is 1.0
<code>exp(x)</code>	exponential function $e^x$	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>fabs(x)</code>	absolute value of $x$	<code>fabs(5.1)</code> is 5.1 <code>fabs(0.0)</code> is 0.0 <code>fabs(-8.76)</code> is 8.76
<code>floor(x)</code>	rounds $x$ to the largest integer not greater than $x$	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>fmod(x, y)</code>	remainder of $x/y$ as a floating-point number	<code>fmod(2.6, 1.2)</code> is 0.2
<code>log(x)</code>	natural logarithm of $x$ (base $e$ )	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of $x$ (base 10)	<code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>pow(x, y)</code>	$x$ raised to power $y$ ( $x^y$ )	<code>pow(2, 7)</code> is 128 <code>pow(9, .5)</code> is 3
<code>sin(x)</code>	trigonometric sine of $x$ ( $x$ in radians)	<code>sin(0.0)</code> is 0
<code>sqrt(x)</code>	square root of $x$ (where $x$ is a non-negative value)	<code>sqrt(9.0)</code> is 3.0
<code>tan(x)</code>	trigonometric tangent of $x$ ( $x$ in radians)	<code>tan(0.0)</code> is 0

## C++11 Additional Math Functions

**11.17** C++11 added dozens of new math functions to the `<cmath>` header. Some were entirely new, and some were additional versions of existing functions for use with arguments of type `float` or `long double`, rather than `double`. The two-argument **hypot** function, for example,

calculates the hypotenuse of a right triangle. C++17 added a three-argument version of `hypot` to calculate the hypotenuse in three-dimensional space. For a complete list of all the `<cmath>` header's functions, see

<https://en.cppreference.com/w/cpp/numeric/math>



or the Section 26.8.1 of the C++ standard document

<https://open-std.org/JTC1/SC22/WG21/docs/papers/>



## 20 C++20—New Mathematical Constants and the `<numbers>` Header

Though C++ has always had common mathematical functions, the C++ standard did not define common mathematical constants. Some C++ implementations defined `M_PI` (for  $\pi$ ) and `M_E` (for  $e$ ) and other mathematical constants via **preprocessor macros**.<sup>1</sup> When the preprocessor executes in those implementations, it replaces these macro names with corresponding `double` floating-point values. Unfortunately, these preprocessor macros were not present in every C++ implementation. C++20's new **`<numbers>` header**<sup>2</sup> standardizes the following mathematical constants commonly used in many scientific and engineering applications:

Constant	Mathematical Expression
numbers::e	$e$
numbers::log2e	$\log_2 e$
numbers::log10e	$\log_{10} e$
numbers::ln2	$\ln 2$
numbers::ln10	$\ln 10$
numbers::pi	$\pi$
numbers::inv_pi	$\frac{1}{\pi}$
numbers::inv_sqrtpi	$\frac{1}{\sqrt{\pi}}$
numbers::sqrt2	$\sqrt{2}$
numbers::sqrt3	$\sqrt{3}$
numbers::inv_sqrt3	$\frac{1}{\sqrt{3}}$
numbers::egamma	Euler-Mascheroni $\gamma$ constant
numbers::phi	$\frac{(1 + \sqrt{5})}{2}$

1. We discuss the preprocessor and macros in [Appendix E](#).

2. <http://wg21.link/p0631r8>.

## 17 C++17 Mathematical Special Functions

For the engineering and scientific communities and other mathematical fields, C++17 added scores of **mathematical special functions**<sup>3</sup> to the `<cmath>` header. You can see the complete list and brief examples of each at:

---

<https://en.cppreference.com/w/cpp/numeric/specia>



3. <http://wg21.link/p0226r1>.

Each mathematical special function in the following table has versions for `float`, `double` and `long double` arguments, respectively:

C++ 17 Mathematical Special Functions	
associated Laguerre polynomials	(incomplete) elliptic integral of the second kind
associated Legendre polynomials	(incomplete) elliptic integral of the third kind
beta function	exponential integral
(complete) elliptic integral of the first kind	Hermite polynomials
(complete) elliptic integral of the second kind	Legendre polynomials
(complete) elliptic integral of the third kind	Laguerre polynomials
regular modified cylindrical Bessel functions	Riemann zeta function
cylindrical Bessel functions (of the first kind)	spherical Bessel functions (of the first kind)
irregular modified cylindrical Bessel functions	spherical associated Legendre functions
cylindrical Neumann functions	spherical Neumann functions
(incomplete) elliptic integral of the first kind	

## 5.4 FUNCTION DEFINITIONS AND FUNCTION PROTOTYPES

In this section, we create a user-defined function called `maximum` that returns the largest of its three `int` arguments.

When the application in Fig. 5.1 executes, the `main` function first reads three integers from the user. Then, the output statement (lines 15–16) calls `maximum`, which is defined in lines 20–34. In line 33, function `maximum` returns the largest value to the point of the `maximum` call (line 16), which is an output statement that displays the result. The sample outputs show that `maximum` determines the largest value regardless of whether it's the first, second or third argument.

---

```
1 // fig05_01.cpp
2 // maximum function with a function prototype.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int maximum(int x, int y, int z); // function prototype
8
9 int main() {
10    cout << "Enter three integer values: ";
11    int int1, int2, int3;
12    cin >> int1 >> int2 >> int3;
13
14    // invoke maximum
15    cout << "The maximum integer value is: "
16    << maximum(int1, int2, int3) << endl;
17 }
18
19 // returns the largest of three integers
20 int maximum(int x, int y, int z) {
21    int maximumValue{x}; // assume x is the largest to start
22
23    // determine whether y is greater than maximumValue
24    if (y > maximumValue) {
25        maximumValue = y; // make y the new maximumValue
26    }
27
28    // determine whether z is greater than maximumValue
29    if (z > maximumValue) {
30        maximumValue = z; // make z the new maximumValue
31    }
32
33    return maximumValue;
34 }
```

```
Enter three integer grades: 86 67 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 86 75
```

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

**Fig. 5.1** maximum function with a function prototype.

### Function `maximum`

Typically, a function definition's first line specifies its return type, function name and **parameter list**, which is enclosed in required parentheses. The parameter list specifies additional information that the function needs to perform its task. The function's first line is also known as the function's **header**. A parameter list may contain zero or more **parameters**, each declared with a type and a name. Two or more parameters are specified using a comma-separated list of parameters. Function `maximum`'s header indicates that the function has three `int` parameters named `x`, `y` and `z`. When you call a function, each parameter receives the corresponding argument's value from the function call.

Function `maximum` first assumes that parameter `x` has the largest value, so line 21 initializes local variable `maximumValue` to parameter `x`'s value. Of course, parameter `y` or `z` may contain the actual largest value, so each of these must be compared with `maximumValue`. Lines 24–26 determine whether `y` is greater than `maximumValue` and, if so, assign `y` to `maximumValue`. Lines 29–31 determine whether `z` is greater than `maximumValue` and, if so, assign `z` to `maximumValue`. At this point, the largest value is in `maximumValue`, so line 33 returns that value to the caller.

### Function Prototype for `maximum`

You must either define a function before using it or declare it, as in line 7:

```
int maximum(int x, int y, int z); // function pr
```

This **function prototype** describes the interface to the `maximum` function without revealing its implementation. A function prototype tells the compiler the function's name, return type and the types of its parameters. Line 7 indicates that `maximum` returns an `int` and requires three `int` parameters to perform its task. The types in the function prototype should be the same as those in the corresponding function definition's header (line 20). The function prototype ends with a required semicolon. We'll see that the names in the function prototype's parameters do not need to match those in the function definition.

### Parameter Names in Function Prototypes

Parameter names in function prototypes are optional (the compiler ignores them), but it's recommended that you use these names for documentation purposes.

### What the Compiler Does with `maximum`'s Function Prototype

When compiling the program, the compiler uses the prototype to

- Ensure that `maximum`'s first line (line 20) matches its prototype (line 7).
- Check that the call to `maximum` (line 16) contains the correct number and types of arguments, and that the types of the arguments are in the correct order (in this case, all the arguments are of the same type).
- Ensure that the value returned by the function can be used correctly in the expression that called the function—for example, a function that does not return a value (declared with the **void return type**) cannot be called on the right side of an assignment.
- Ensure that each argument is consistent with the type of the corresponding parameter—for example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not a string like "hello". If the arguments

passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types. Section 5.6 discusses this conversion process and what happens if the conversion is not allowed.

Compilation errors occur if the function prototype, header and calls do not all agree in the number, type and order of arguments and parameters, and in the return type. For calls, the compiler checks whether the function's return value (if any) can be used where the function was called.

### Returning Control from a Function to Its Caller

When a program calls a function, the function performs its task, then returns control (and possibly a value) to the point where the function was called. In a function that does not return a result (i.e., it has a `void` return type), control returns when the program reaches the function-ending right brace. A function can explicitly return control (and no result) to the caller by executing

```
return;
```

anywhere in the function's body.

## 5.5 ORDER OF EVALUATION OF A FUNCTION'S ARGUMENTS

Multiple parameters are specified in function prototypes, function headers and function calls as comma-separated lists. The commas in line 16 of Fig. 5.1 that separate function `maximum`'s arguments are not comma operators. The comma operator guarantees that its operands evaluate left-to-right. The order of evaluation of a function's arguments, however, is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders.

Sometimes when a function's arguments are expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes

between compilers, the argument values passed to the function could vary, causing subtle logic errors.

If you have doubts about the order of evaluation of a function's arguments and whether the order would affect the values passed to the function, assign the arguments to variables before the call, then pass those variables as arguments to the function.

## 5.6 FUNCTION-PROTOTYPE AND ARGUMENT-COERCION NOTES

A function prototype is required unless the function is defined before it's used. When you use a standard library function like `sqrt`, you do not have access to the function's definition, so it cannot be defined in your code before you call the function. Instead, you must include its corresponding header (in this case, `<cmath>`), which contains the function's prototype.

If a function is defined before it's called, then its definition also serves as the function's prototype, so a separate prototype is unnecessary. If a function is called before it's defined, and that function does not have a function prototype, a compilation error occurs.

Always provide function prototypes, even though it's possible to omit them when functions are defined before they're used. Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).

### 5.6.1 Function Signatures and Function Prototypes

A function's name and its argument types together are known as the **function signature** or simply the **signature**. The function's return type is not part of the function signature. The scope of a function is the region of a program in which the function is known and accessible. Functions in the same scope must have unique signatures. We'll say more about scope in Section 5.11.

In Fig. 5.1, if the function prototype in line 7 had been written

```
void maximum(int x, int y, int z);
```

the compiler would report an error, because the `void` return type in the function prototype would differ from the `int` return type in the function header. Similarly, such a prototype would cause the statement

```
cout << maximum(6, 7, 0);
```

to generate a compilation error because that statement depends on `maximum` returning a value to be displayed. Function prototypes help you find many types of errors at compile-time, which is always better than finding them at run time.

### 5.6.2 Argument Coercion

An important feature of function prototypes is **argument coercion**—i.e., forcing arguments to the appropriate types specified by the parameter declarations. For example, a program can call a function with an integer argument, even though the function prototype specifies a `double` parameter—the function will still work correctly, provided this is not a narrowing conversion (discussed in Section 3.8.3). A compilation error occurs if the arguments in a function call cannot be implicitly converted to the expected types specified in the function’s prototype.

### 5.6.3 Argument-Promotion Rules and Implicit Conversions<sup>4</sup>

<sup>4</sup>. Promotions and conversions are complex topics discussed in Sections 7.3 and 7.4 of the C++ standard document. “Working Draft, Standard for Programming Language C.” Accessed May 11, 2020. <http://open-std.org/JTC1/SC22/WG21/docs/papers/2020/n4861.pdf>.

**11** Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called. These conversions occur as specified by C++’s **promotion rules**, which indicate the implicit conversions allowed between fundamental types. An `int` can

be converted to a `double`. A `double` can also be converted to an `int`, but this narrowing conversion truncates the `double`'s fractional part—recall from Section 3.8.3 that C++11 list initializers do not allow narrowing conversions. Keep in mind that `double` variables can hold numbers of much greater magnitude than `int` variables, so the loss of data in a narrowing conversion can be considerable.

Values might also be modified when converting large integer types to small integer types (e.g., `long` to `short`), signed to `unsigned`, or `unsigned` to signed. Variables of **`unsigned`** integer types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types. The `unsigned` types are used primarily for bit manipulation ([Chapter 22](#)). They should not be used to ensure that a value is non-negative.<sup>5</sup>

<sup>5</sup>. C++ Core Guidelines. Accessed May 11, 2020.  
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-expects>.

The promotion rules also apply to expressions containing values of two or more data types—that is, **`mixed-type expressions`**. Each value's type in a mixed-type expression is promoted to the expression's “highest” type (actually a temporary copy of each value is created and used—the original values remain unchanged). The following table lists the arithmetic data types in order from “highest type” to “lowest type.”

## Data types

<code>long double</code>	
<code>double</code>	
<code>float</code>	
<code>unsigned long long int</code>	(synonymous with <code>unsigned long long</code> )
<code>long long int</code>	(synonymous with <code>long long</code> )
<code>unsigned long int</code>	(synonymous with <code>unsigned long</code> )
<code>long int</code>	(synonymous with <code>long</code> )
<code>unsigned int</code>	(synonymous with <code>unsigned</code> )
<code>int</code>	
<code>unsigned short int</code>	(synonymous with <code>unsigned short</code> )
<code>short int</code>	(synonymous with <code>short</code> )
<code>unsigned char</code>	
<code>char</code> and <code>signed char</code>	
<code>bool</code>	

## Conversions Can Result in Incorrect Values

Converting values to lower fundamental types can cause errors due to narrowing conversions. Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type (some compilers will issue a warning in this case) or by using a cast operator. Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types. If a `square` function with an `int` parameter is called with a `double` argument, the argument is converted to `int` (a lower type and thus a narrowing conversion), and `square` could return an incorrect value. For example, `square(4.5)` would return 16, not 20.25. Some compilers warn you about the

narrowing conversion. For example, Microsoft Visual C++ issues the warning,

```
'argument': conversion from 'double' to 'int', p
```

## Narrowing Conversions with the Guidelines Support Library

If you must perform an explicit narrowing conversion, the C++ Core Guidelines recommend using a **narrow\_cast operator**<sup>6</sup> from the **Guidelines Support Library (GSL)**—we'll use this in Figs. 5.5 and 5.6. This library has several implementations. Microsoft provides an open-source version that has been tested on numerous platform/compiler combinations, including our three preferred compilers and platforms. You can download their GSL implementation from:

<https://github.com/Microsoft/GSL>

<sup>6</sup>. C++ Core Guidelines. Accessed May 10, 2020.  
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing>.

For your convenience, we provided the GSL library with this book's code examples in the subfolder **libraries/GSL**.

The GSL is a header-only library, so you can use it in your programs simply by including the header "gsl/gsl". You must point your compiler to the **GSL** folder's **include** sub-folder, so the compiler knows where to find the include file, as you did when you used class **BigNumber** at the end of Section 3.12. The following statement uses the **narrow\_cast** operator (from namespace **gsl**) to convert the double value 7.5 to the **int** value 7:

```
gsl::narrow_cast<int>(7.5)
```

As with the other named cast operators, like **static\_cast**, the value in parentheses is converted to the type in angle brackets, **<>**.

## 5.7 C++ STANDARD LIBRARY HEADERS

The C++ Standard Library is divided into many portions, each with its own header. The headers contain the function prototypes for the related functions that form each portion of the library. The headers also contain definitions of various class types and functions, as well as constants needed by those functions. A header “instructs” the compiler on how to interface with library and user-written components.

The following table lists some common C++ Standard Library headers, many of which are discussed later in this book. The term “macro” that’s used several times is discussed in detail in [Appendix E, Preprocessor](#). For a complete list of the 96 C++20 standard library headers, visit

---

<https://en.cppreference.com/w/cpp/header>

On that page, you’ll see approximately three dozen additional headers that are marked as either deprecated or removed. Deprecated headers are ones you should no longer use, and removed headers are no longer included in the C++ standard library.

Standard Library header	Explanation
<code>&lt;iostream&gt;</code>	Function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output: A Deeper Look.
<code>&lt;iomanip&gt;</code>	Function prototypes for stream manipulators that format streams of data. This header is first used in Section 3.7 and is discussed in more detail in Chapter 15, Stream Input/Output: A Deeper Look.
<code>&lt;cmath&gt;</code>	Function prototypes for math library functions (Section 5.3).
<code>&lt;cstdlib&gt;</code>	Function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 5.8; Chapter 14, Operator Overloading; Class <code>string</code> ; Chapter 18, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and <code>structs</code> ; and Appendix F, C Legacy Code Topics.
11 20	<code>&lt;ctime&gt;, &lt;chrono&gt;</code> Function prototypes and types for manipulating the time and date. <code>&lt;ctime&gt;</code> is used in Section 5.8. <code>&lt;chrono&gt;</code> was introduced in C++11 and enhanced with many more features in C++20
11	<code>&lt;array&gt;, &lt;vector&gt;, &lt;list&gt;, &lt;forward_list&gt;, &lt;deque&gt;, &lt;queue&gt;, &lt;stack&gt;, &lt;map&gt;, &lt;unordered_map&gt;, &lt;unordered_set&gt;, &lt;set&gt;, &lt;bitset&gt;</code> These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <code>&lt;vector&gt;</code> header is first introduced in Chapter 6, Class Templates array and vector; Catching Exceptions. We discuss all these headers in Chapter 16, Standard Library Containers and Iterators. <code>&lt;array&gt;, &lt;forward_list&gt;, &lt;unordered_map&gt;</code> and <code>&lt;unordered_set&gt;</code> were all introduced in C++11.
	<code>&lt;cctype&gt;</code> Function prototypes for functions that test characters for certain properties (such as whether the character is a digit or punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and <code>structs</code> .

<code>&lt;cstring&gt;</code>	Function prototypes for C-style string-processing functions.
<code>&lt;typeinfo&gt;</code>	Classes for runtime type identification (determining data types at execution time). This header is discussed in Section 13.9.
<code>&lt;exception&gt;, &lt;stdexcept&gt;</code>	Classes for exception handling (discussed in Chapter 18, Exception Handling: A Deeper Look).
<code>&lt;memory&gt;</code>	Classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 18, Exception Handling: A Deeper Look.
<code>&lt;fstream&gt;</code>	Function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 9, File Processing and String Stream Processing).
<code>&lt;string&gt;</code>	Definition of class <code>string</code> from the C++ Standard Library (discussed in Chapter 8, Class <code>string</code> ).
<code>&lt;iostream&gt;</code>	Function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 8, Class <code>string</code> and String Stream Processing).
<code>&lt;functional&gt;</code>	Classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 17.
<code>&lt;iterator&gt;</code>	Classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 16.
<code>&lt;algorithm&gt;</code>	Functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 16.
<code>&lt;cassert&gt;</code>	Macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.

	<cfloat>	Floating-point size limits of the system.
	<climits>	Integral size limits of the system.
	<cstdio>	Function prototypes for C-style standard input/output library functions.
	<locale>	Classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
	<limits>	Classes for defining the numerical data type limits on each computer platform—this is C++’s version of <climits> and <cfloat>.
	<utility>	Classes and functions that are used by many C++ Standard Library headers.
11	<thread>, <mutex>,	Capabilities added in C++11 and C++14 for multithreaded application development so your applications can take advantage of today’s multicore processors (discussed in the Concurrency chapter).
14	<shared_mutex>,	
	<future>,	
	<condition_variable>	
17	<i>Some Key C++17 New Headers</i>	
	<any>	A template for holding a value of any type (discussed in Chapter 16, Standard Library Containers and Iterators).
	<optional>	A template to represent an object that may or may not have a value (discussed in Chapter 16, Standard Library Containers and Iterators).
	<execution>	Features used with the Standard Template Library’s parallel algorithms (discussed in the Concurrency chapter).
	<filesystem>	Capabilities for interacting with the local filesystem’s files and folders (discussed in Chapter 9).

<concepts>	Capabilities for constraining the types that can be used with templates.
<coroutine>	Capabilities for asynchronous programming with coroutines (discussed in the Concurrency chapter).
<compare>	Support for the new three-way comparison operator <code>&lt;=&gt;</code> (discussed in Chapter 14, Operator Overloading; Class <code>string</code> ).
<format>	New concise and powerful text-formatting capabilities (discussed throughout the book).
<ranges>	Capabilities that support functional-style programming (discussed in Chapter 6 and Chapter 16, Standard Library Containers and Iterators).
<span>	Capabilities for creating views into contiguous sequences of objects.
<bit>	Standardized bit manipulation operations.
<stop_token>, <semaphore>, <latch>, <barrier>	Additional capabilities that support the multithreaded application-development features added in C++11 and C++14.

## 5.8 CASE STUDY: RANDOM-NUMBER GENERATION

We now take a brief and hopefully entertaining diversion into a popular programming application, namely simulation and game playing. In this and the next section, we develop a game-playing program that includes multiple functions.

### The `rand` Function

The element of chance can be introduced into computer applications by using the C++ Standard Library function **rand**. Consider the following statement:

```
i = rand();
```

The function `rand` generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<cstdlib>` header). You can determine the value of `RAND_MAX` for your

system simply by displaying the constant. If `rand` truly produces integers at random, every number between 0 and `RAND_MAX` has an equal chance (or probability) of being chosen each time `rand` is called.

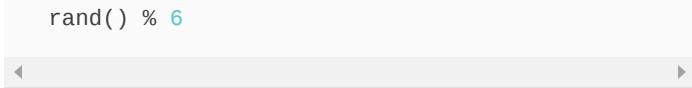
The range of values produced directly by the function `rand` often is different than what a specific application requires. For example, a program that simulates coin tossing might need only 0 for “heads” and 1 for “tails.” A program that simulates rolling a six-sided die would require random integers in the range 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1 through 4.

### 5.8.1 Rolling a Six-Sided Die

To demonstrate `rand`, Fig. 5.2 simulates ten rolls of a six-sided die and displays the value of each roll. The function prototype for the `rand` function is in `<cstdlib>`. To produce integers in the range 0 to 5, we use the remainder operator (%) with `rand` as follows:

---

```
rand() % 6
```



This is called **scaling**. The number 6 is called the **scaling factor**. We then **shift** the range of numbers produced by adding 1 to our previous result. Figure 5.2 confirms that the results are in the range 1 to 6. If you execute this program more than once, you’ll see that it produces the same “random” values each time. We’ll show how to fix this in Figure 5.4.

---

```
1 // fig05_02.cpp
2 // Shifted, scaled integers produced by 1 + rand() % 6.
3 #include <iostream>
4 #include <cstdlib> // contains function prototype for rand
5 using namespace std;
6
7 int main() {
8     for (int counter[1]; counter <= 10; ++counter) {
9         // pick random number from 1 to 6 and output it
10        cout << (1 + rand() % 6) << " ";
11    }
12
13    cout << endl;
14 }
```

```
6 6 5 5 6 5 1 1 5 3
```

**Fig. 5.2** Shifted, scaled integers produced by `1 + rand() % 6`. (Part 1 of 2.)

### 5.8.2 Rolling a Six-Sided Die 60,000,000 Times

To show that values produced by `rand` occur with approximately equal likelihood, Fig. 5.3 simulates 60,000,000 rolls of a die.<sup>7</sup> Each integer in the range 1 to 6 should appear approximately 10,000,000 times (one-sixth of the rolls). The program’s output confirms this. The `face` variable’s definition in the `switch`’s initializer (line 19) is preceded by `const`. This is a good practice for any variable that should not change once it’s initialized—this enables the compiler to report errors if you accidentally modify the variable.

7. When co-author Harvey Deitel first implemented this example for his classes in 1976, he performed only 600 die rolls—6000 would have taken too long. On our system, this program took approximately five seconds to complete 60,000,000 die rolls! 600,000,000 die rolls took approximately one minute. The die rolls occur sequentially. In our concurrency chapter, we’ll explore how to parallelize this application to take advantage of today’s multi-core computers.

---

```
1 // fig05_03.cpp
2 // Rolling a six-sided die 60,000,000 times.
3 #include <iostream>
4 #include <iomanip>
5 #include <cstdlib> // contains function prototype for rand
6 using namespace std;
7
8 int main() {
9     int frequency1{0}; // count of 1s rolled
10    int frequency2{0}; // count of 2s rolled
11    int frequency3{0}; // count of 3s rolled
12    int frequency4{0}; // count of 4s rolled
13    int frequency5{0}; // count of 5s rolled
14    int frequency6{0}; // count of 6s rolled
15
16    // summarize results of 60,000,000 rolls of a die
17    for (int roll{1}; roll <= 60'000'000; ++roll) {
18        // determine roll value 1-6 and increment appropriate counter
19        switch (const int face{1 + rand() % 6}; face) {
20            case 1:
21                ++frequency1; // increment the 1s counter
22                break;
23            case 2:
24                ++frequency2; // increment the 2s counter
25                break;
26            case 3:
27                ++frequency3; // increment the 3s counter
28                break;
29            case 4:
30                ++frequency4; // increment the 4s counter
```

```

31         break;
32     case 5:
33         ++frequency5; // increment the 5s counter
34         break;
35     case 6:
36         ++frequency6; // increment the 6s counter
37         break;
38     default: // invalid value
39         cout << "Program should never get here!";
40     }
41 }
42
43 cout << "Face" << setw(13) << "Frequency" << endl; // output headers
44 cout << " 1" << setw(13) << frequency1
45 << "\n 2" << setw(13) << frequency2
46 << "\n 3" << setw(13) << frequency3
47 << "\n 4" << setw(13) << frequency4
48 << "\n 5" << setw(13) << frequency5
49 << "\n 6" << setw(13) << frequency6 << endl;
50 }

```

Face	Frequency
1	9999294
2	10002929
3	9995360
4	10000409
5	10005206
6	9996802

**Fig. 5.3** Rolling a six-sided die 60,000,000 times. (Part 1 of 2.)

As the output shows, we can simulate rolling a six-sided die by scaling and shifting the values `rand` produces. The default case (lines 38–39) in the `switch` should never execute because the switch’s controlling expression (`face`) always has values in the range 1–6. After we study arrays in Chapter 6, we show how to replace the entire switch in Fig. 5.3 elegantly with a single-line statement. Many programmers provide a

default case in every `switch` statement to catch errors even if they feel confident that their programs are error-free.

## 14 C++14 Digit Separators for Numeric Literals

Before C++14, you'd represent the integer value 60,000,000 as `60000000` in a program. Typing numeric literals with many digits can be error-prone. To make numeric literals more readable and help reduce errors, you can use C++14's **digit separator**' (a single-quote character) between groups of digits—`60'000'000` (line 17) represents the integer value 60,000,000. You might wonder why single-quote characters are used rather than commas. If we use `60,000,000` in line 17, C++ treats the commas as comma operators, and the value of `60,000,000` would be the rightmost expression (`000`). The loop-continuation condition would immediately be false—a logic error in this program.

### 5.8.3 Randomizing the Random-Number Generator with `srand`

Executing the program of Fig. 5.2 again produces

```
6 6 5 5 6 5 1 1 5 3
```

This is the same sequence of values shown in Fig. 5.2. How can these be random numbers?

Function `rand` actually generates **pseudorandom numbers**. Repeatedly calling `rand` produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program executes. When debugging a simulation program, random-number repeatability is essential for proving that corrections to the program work properly. Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called **randomizing** and can be accomplished with the C++ Standard Library function `srand` from the header `<cstdlib>`. Function `srand` takes an `unsigned` integer argument and **seeds** the `rand` function to

produce a different sequence of random numbers for each execution.

### Using `srand`

Figure 5.4 demonstrates function `srand`. The program produces a different sequence of random numbers each time it executes, provided that the user enters a different seed. We used the same seed in the first and third sample outputs, so the same series of 10 numbers is displayed in each of those outputs.



**Security** For security, ensure that your program seeds the random-number generator differently (and only once) each time the program executes; otherwise, an attacker would easily be able to determine the sequence of pseudorandom numbers that would be produced.

---

```
1 // fig05_04.cpp
2 // Randomizing the die-rolling program.
3 #include <iostream>
4 #include <iomanip>
5 #include <cstdlib> // contains prototypes for functions srand and rand
6 using namespace std;
7
8 int main() {
9     int seed{0}; // stores the seed entered by the user
10
11    cout << "Enter seed: ";
12    cin >> seed;
13    srand(seed); // seed random number generator
14
15    // Loop 10 times
16    for (int counter{1}; counter <= 10; ++counter) {
17        // pick random number from 1 to 6 and output it
18        cout << (1 + rand() % 6) << " ";
19    }
20
21    cout << endl;
22 }
```

```
Enter seed: 67  
6 1 4 6 2 1 6 1 6 4
```

```
Enter seed: 432  
4 6 3 1 6 3 1 5 4 2
```

```
Enter seed: 67  
6 1 4 6 2 1 6 1 6 4
```

**Fig. 5.4** Randomizing the die-rolling program. (Part 1 of 2.)

#### 5.8.4 Seeding the Random-Number Generator with the Current Time

To randomize without having to enter a seed each time, we can use a statement like

```
    srand(gsl::narrow_cast<unsigned int>(time(0)));
```

This causes the computer to read its clock to obtain the value for the seed. Function **time** (with the argument **0** as written in the preceding statement) typically returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT). This value (which is of type **time\_t**) is converted to **unsigned int** and used as the seed to the random-number generator. The **narrow\_cast** (from the Guidelines Support Library) in the preceding statement eliminates a compiler warning that's issued if you pass a **time\_t** value to a function that expects an **unsigned int**.<sup>8</sup> The function prototype for **time** is in **<ctime>**.

8. Our code originally used a **static\_cast** rather than a **narrow\_cast**. The C++ Core Guidelines checker in Microsoft Visual C++ reported that narrowing conversions should be performed with **narrow\_cast** operators.

#### 5.8.5 Scaling and Shifting Random Numbers

Previously, we simulated rolling a six-sided die with the statement

```
int face{1 + rand() % 6};
```

which always assigns an integer (at random) to variable `face` in the range  $1 \leq \text{face} \leq 6$ . The width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale `rand` with the remainder operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that is added to the expression `rand % 6`. We can generalize this result as

```
int variableName{shiftingValue + rand() %  
scalingFactor};
```

where the *shiftingValue* is equal to the first number in the desired range of consecutive integers, and the *scalingFactor* is equal to the width of the desired range of consecutive integers.

## 5.9 CASE STUDY: GAME OF CHANCE; INTRODUCING SCOPED ENUMS

One of the most popular games of chance is a dice game known as “craps,” which is played in casinos and back alleys worldwide. The rules of the game are straightforward:

*A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, the player wins. If the sum is 2, 3 or 12 on the first roll (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player’s “point.” To win, you must keep rolling the dice until you “make your point.”*

*The player loses by rolling a 7 before making the point.*

In the rules, notice that the player must roll two dice on the first roll and all subsequent rolls. We will define a `rollDice` function to roll the dice and compute and display their sum. The function will be defined once, but may be called multiple times—once for the game’s first roll and possibly many more times if the player does not win or lose on the first roll. Below are the outputs of several sample executions showing:

- winning on the first roll by rolling a 7,
- losing on the first roll by rolling a 12,
- winning on a subsequent roll by “making the point” before rolling a 7, and
- losing on a subsequent roll by rolling a 7 before “making the point.”

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
```

```
Player rolled 4 + 3 = 7  
Player loses
```

## Implementing the Game

The craps program (Fig. 5.5) simulates the game using two functions—`main` and `roll-Dice`—and the `switch`, `while`, `if...else`, nested `if...else` and nested `if` statements. Function `rollDice`'s prototype (line 9) indicates that the function takes no arguments (empty parentheses) and returns an `int` (the sum of the dice).

```
1 // fig05_05.cpp  
2 // Craps simulation.  
3 #include <iostream>  
4 #include <cstdlib> // contains prototypes for functions srand and rand  
5 #include <ctime> // contains prototype for function time  
6 #include "gs1/gs1" // Guidelines Support Library for narrow_cast  
7 using namespace std;  
8  
9 int rollDice(); // rolls dice, calculates and displays sum  
10  
11 int main() {
```

**Fig. 5.5** Craps simulation.

## C++11: Scoped enums

The player may win or lose on the first roll or any subsequent roll. The program tracks this with the variable `gameStatus`, which line 19 declares to be of the new type `Status`. Line 13 declares a user-defined type called a **scoped enumeration** which is introduced by the keywords `enum class`, followed by a type name (`Status`) and a set of identifiers representing integer constants:

```
12 // scoped enumeration with constants that  
represent the game status  
13 enum class Status {keepRolling, won, lost}; //  
all caps in constants  
14  
15 // randomize random number generator using
```

```
current time
16   srand(gsl::narrow_cast<unsigned int>(time(0)));
17
18   int myPoint{0}; // point if no win or loss on
first roll
19   Status gameStatus{Status::keepRolling}; // game
is not over
20
```

---

The underlying values of these **enumeration constants** are of type `int`, start at `0` and increment by `1`, by default (you’ll soon see how to change this). In the `Status` enumeration, the constant `keepRolling` has the value `0`, `won` has the value `1`, and `lost` has the value `2`. The identifiers in an `enum class` must be unique, but separate enumeration constants can have the same value. Variables of user-defined type `Status` can be assigned only the constants declared in the enumeration.

By convention, you should capitalize the first letter of an `enum class`’s name and the first letter of each subsequent word in a multi-word `enum class` name (e.g., `ProductCode`). The constants in an `enum class` should use the same naming conventions as variables.<sup>9,10</sup>

<sup>9</sup> C++ Core Guidelines. Accessed May 11, 2020.  
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps>.

<sup>10</sup> In legacy C++ code, you’ll commonly see `enum` constants in all uppercase letters—that practice is now deprecated.

To reference a scoped `enum` constant, qualify the constant with the scoped `enum`’s type name (in this example, `Status`) and the scope-resolution operator (`::`), as shown in line 19, which initializes `gameStatus` to `Status::keepRolling`. For a win, the program sets `gameStatus` to `Status::won`. For a loss, the program sets `gameStatus` to `Status::lost`.

## Winning or Losing on the First Roll

The following `switch` determines whether the player wins or loses on the first roll:

---

```
21 // determine game status and point (if needed)
based on first roll
```

```
22     switch (const int sumOfDice{rollDice()});
23     sumOfDice) {
24         case 7: // win with 7 on first roll
25         case 11: // win with 11 on first roll
26             gameStatus = Status::won;
27             break;
28         case 2: // lose with 2 on first roll
29         case 3: // lose with 3 on first roll
30         case 12: // lose with 12 on first roll
31             gameStatus = Status::lost;
32             break;
33         default: // did not win or lose, so remember
34             myPoint = sumOfDice; // remember the point
35             cout << "Point is " << myPoint << endl;
36             break; // optional at end of switch
37 }
```

---

**17** The switch's initializer (line 22) creates the variable `sumOfDice` and initializes it by calling function `rollDice`. If the roll is 7 or 11, line 25 sets `gameStatus` to `Status::won`. If the roll is 2, 3, or 12, line 30 sets `gameStatus` to `Status::lost`. For other values, `gameStatus` remains unchanged (`Status::keepRolling`), line 33 saves `sumOfDice` in `myPoint`, and line 34 displays the `myPoint`.

## Continuing to Roll

After the first roll, if `gameStatus` is `Status::keepRolling`, execution proceeds with the following `while` statement:

---

```
38     // while game is not complete
39     while (Status::keepRolling == gameStatus) { // not won or lost
40         // roll dice again and determine game status
41         if (const int sumOfDice{rollDice()});
42         sumOfDice == myPoint) {
43             gameStatus = Status::won;
44         }
45         else {
46             if (sumOfDice == 7) { // lose by rolling
47                 gameStatus = Status::lost;
48             }
49         }
50     }
51 }
```

```
48     }
49 }
50
```

---

**17** In each iteration of the `while`, the `if` statement's initializer (line 41) calls `rollDice` to produce a new `sumOfDice`. If `sumOfDice` matches `myPoint`, the program sets `gameStatus` to `Status::won` (line 42), and the loop terminates. If `sumOfDice` is 7, the program sets `gameStatus` to `Status::lost` (line 46), and the loop terminates. Otherwise, the loop continues executing.

### Displaying Whether the Player Won or Lost

When the preceding loop terminates, the program proceeds to the following `if...else` statement, which prints "Player wins" if `gameStatus` is `Status::won` or "Player loses" if `gameStatus` is `Status::lost`:

```
51 // display won or lost message
52 if (Status::won == gameStatus) {
53     cout << "Player wins" << endl;
54 }
55 else {
56     cout << "Player loses" << endl;
57 }
58 }
59
```

---

### Function `rollDice`

To roll the dice, function `rollDice` uses the die-rolling calculation shown previously to initialize variables `die1` and `die2`, then calculates their `sum`. Lines 67–68 display `die1`'s, `die2`'s and `sum`'s values before line 69 returns `sum`.

```
60 // roll dice, calculate sum and display results
61 int rollDice() {
62     const int die1{1 + rand() % 6}; // first die
63     const int die2{1 + rand() % 6}; // second die
64     const int sum{die1 + die2}; // compute sum of
65     // die values
66 // display results of this roll
```

```
67     cout << "Player rolled " << die1 << " + " <<
die2
68         << " = " << sum << endl;
69     return sum;
70 }
```

---

## Additional Notes Regarding Scoped enum's

Qualifying an `enum class`'s constant with its type name and `::` explicitly identifies the constant as being in the scope of the specified `enum class`. If another `enum class` contains the same identifier, it's always clear which constant is being used because the type name and `::` are required. In general, you should use unique values for an `enum`'s constants to help prevent hard-to-find logic errors.

Another popular scoped enumeration is

```
enum class Months {jan = 1, feb, mar, apr, may,
sep, oct, nov, dec};
```



which creates user-defined `enum class` type `Months` with enumeration constants representing the months of the year. The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12. Any enumeration constant can be assigned an integer value in the enumeration definition. Subsequent enumeration constants each have a value 1 higher than the preceding constant until the next explicit setting.

## Enumeration Types Before C++11

Enumerations also can be defined with the keyword `enum` followed by a type name and a set of integer constants represented by identifiers, as in

```
enum Status {keepRolling, won, lost};
```



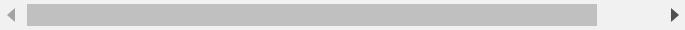
The constants in such an `enum` are unscoped—you can refer to them simply by their names `keepRolling`, `won` and `lost`. If two or more unscoped `enums` contain constants with

the same names, this can lead to naming conflicts and compilation errors.

## 11 C++11—Specifying the Type of an `enum`'s Constants

An enumeration's constants have integer values. An unscoped `enum`'s underlying type depends on its constants' values and is guaranteed to be large enough to store its constants' values. A scoped `enum`'s underlying integral type is `int`, but you can specify a different type by following the type name with a colon (:) and the integral type. For example, we can specify that the constants in the `enum class Status` should have type `long`, as in

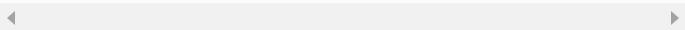
```
enum class Status : long {keepRolling, won, lost
```



## 20 C++20—`using enum` Declaration

If the type of an `enum class`'s constants is obvious, based on the context in which they're used—such as in our craps example—C++20's **`using enum` declaration**<sup>11,12</sup> allows you to reference an `enum class`'s constants without the type name and scope-resolution operator (::). For example, adding the following statement after the `enum class` declaration:

```
using enum Status;
```

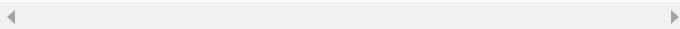


11. <http://wg21.link/p1099r5>.

12. At the time of this writing, this feature works only in Microsoft's Visual C++ compiler.

would allow the rest of the program to use `keepRolling`, `won` and `lost`, rather than `Status::keepRolling`, `Status::won` and `Status::lost`, respectively. You also may use an individual `enum class` constant with a declaration of the form

```
using enum Status::keepRolling;
```



This would allow your code to use `keepRolling` without the `Status::` qualifier.

## 11 5.10 C++11'S MORE SECURE NONDETERMINISTIC RANDOM NUMBERS



Security Function `rand`—which was inherited into C++ from the C Standard Library—does not have “good statistical properties” and can be predictable.<sup>13</sup> This makes programs that use `rand` less secure. C++11 provides a 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. These capabilities are located in the C++ Standard Library’s `<random>` header.

<sup>13</sup>. “Do not use the `rand()` function for generating pseudorandom numbers.”

Accessed May 9, 2020.

<https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers>.

Random-number generation is a sophisticated topic for which mathematicians have developed many algorithms with different statistical properties. For flexibility based on how random numbers are used in programs, C++11 provides many classes that represent various **random-number generation engines and distributions**. An engine implements a random-number generation algorithm that produces pseudorandom numbers. A distribution controls the range of values produced by an engine, the value’s types (e.g., `int`, `double`, etc.) and the value’s statistical properties. We’ll use the default random-number generation engine—**default\_random\_engine**—and a **uniform\_int\_distribution**, which evenly distributes pseudorandom integers over a specified value

range. The default range is from 0 to the maximum value of an `int` on your platform.

### **Rolling a Six-Sided Die**

Figure 5.6 uses the `default_random_engine` and the `uniform_int_distribution` to roll a six-sided die. Line 14 creates a `default_random_engine` object named `engine`. Its constructor argument seeds the random-number generation engine with the current time. If you don't pass a value to the constructor, the default seed will be used, and the program will produce the same sequence of numbers each time it executes—this is useful for testing purposes. Line 15 creates `randomInt`—a `uniform_int_distribution` object that produces `int` values (specified by `<int>`) in the range 1 to 6 (specified by the initializer `{1, 6}`). The expression `randomInt(engine)` (line 20) returns one `int` in the range 1 to 6.

---

```
1 // fig05_06.cpp
2 // Using a C++11 random-number generation engine and distribution
3 // to roll a six-sided die more securely.
```

```

4 #include <iostream>
5 #include <iomanip>
6 #include <random> // contains C++11 random number generation features
7 #include <ctime>
8 #include "gs1/gs1"
9 using namespace std;
10
11 int main() {
12     // use the default random-number generation engine to
13     // produce uniformly distributed pseudorandom int values from 1 to 6
14     default_random_engine engine(gs1::narrow_cast<unsigned int>(time(0)));
15     const uniform_int_distribution<int> randomInt{1, 6};
16
17     // loop 10 times
18     for (int counter{1}; counter <= 10; ++counter) {
19         // pick random number from 1 to 6 and output it
20         cout << setw(10) << randomInt(engine);
21     }
22
23     cout << endl;
24 }
```

2 1 2 3 5 6 1 5 6 4

**Fig. 5.6** Using a C++11 random-number generation engine and distribution to roll a six-sided die more securely. (Part 1 of 2.)

The notation `<int>` in line 15 indicates that `uniform_int_distribution` is a class template. In this case, any integer type can be specified in the angle brackets (`<` and `>`). In Chapter 19, we discuss how to create class templates, and various other chapters show how to use existing class templates from the C++ Standard Library. For now, you can use class template `uniform_int_distribution` by mimicking the syntax shown in the example.

## 5.11 SCOPE RULES

The portion of a program where an identifier can be used is known as its scope. For example, when we declare a local variable in a block, it can be referenced only

- from the point of its declaration in that block and
- in nested blocks that appear within that block after the variable's declaration.

This section discusses block scope and global namespace scope. Parameter names in function prototypes have **function-prototype scope**—they're known only in the prototype in which they appear. Later we'll see other scopes, including **class scope** in Chapter 11, and **function scope** and **namespace scope** in Chapter 23.

## Block Scope

Identifiers declared inside a block have **block scope**, which begins at the identifier's declaration and ends at the terminating right brace (}) of the enclosing block. Local variables have block scope, as do function parameters (even though they're declared outside the block's braces). Any block can contain variable declarations. When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the one in the outer block is “hidden” until the inner block terminates. The inner block “sees” its own local variable's value and not that of the enclosing block's identically named variable.

Accidentally using the same name for an identifier in an inner block that's used for an identifier in an outer block when, in fact, you want the identifier in the outer block to be active for the duration of the inner block, is typically a logic error. Avoid variable names in inner scopes that hide names in outer scopes. Most compilers will warn you about this.

Local variables also may be declared **static**. Such variables also have block scope, but unlike other local variables, a **static** local variable retains its value when the function returns to its caller. The next time the function is called, the **static** local variable contains the value it had when the

function last completed execution. The following statement declares **static** local variable **count** and initializes to 1:

```
static int count{1};
```

By default, all **static** local variables of numeric types are initialized to zero. The following statement declares **static** local variable **count** and initializes it to 0:

```
static int count;
```

Default initialization of non-fundamental-type variables depends on the type—for example, a **string**’s default value is the empty string (""). We’ll say more about default initialization in later chapters.

## Global Namespace Scope

An identifier declared outside any function or class has **global namespace scope**. Such an identifier is “known” to all functions after its declaration in the source-code file. Function definitions, function prototypes placed outside a function, class definitions and global variables all have global namespace scope. **Global variables** are created by placing variable declarations outside any class or function definition. Such variables retain their values throughout a program’s execution.

Declaring a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. Except for truly global resources such as **cin** and **cout**, you should avoid global variables. This is an example of the **principle of least privilege**, which is fundamental to good software engineering. It states that code should be granted *only* the amount of privilege and access that it needs to accomplish its designated task, but no more. An example of this is the scope of a local variable, which should not be visible when it’s not needed. A local variable is created when

the function is called, used by that function while it executes then goes away when the function returns. The principle of least privilege makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values that should not be accessible to it.

In general, variables should be declared in the narrowest scope in which they need to be accessed. Variables used only in a particular function should be declared as local variables in that function rather than as global variables.

### Scope Demonstration

Figure 5.7 demonstrates scoping issues with global variables, local variables and `static` local variables. We show portions of the program with their corresponding outputs for discussion purposes. Line 10 declares and initializes global variable `x` to 1. This global variable is hidden in any block (or function) that declares a variable named `x`.

---

```
1 // fig05_07.cpp
2 // Scoping example.
3 #include <iostream>
4 using namespace std;
5
6 void useLocal(); // function prototype
7 void useStaticLocal(); // function prototype
8 void useGlobal(); // function prototype
9
10 int x[1]; // global variable
11
```

---

**Fig. 5.7** Scoping example.

### Function `main`

In `main`, line 13 displays the value of global variable `x`. Line 15 initializes local variable `x` to 5. Line 17 outputs this variable to show that the global `x` is hidden in `main`. Next, lines 19–23 define a new block in `main` in which another local variable `x` is initialized to 7 (line 20). Line 22 outputs

this variable to show that it hides `x` in the outer block of `main` as well as the global `x`. When the block exits, the variable `x` with value 7 is destroyed automatically. Next, line 25 outputs the local variable `x` in the outer block of `main` to show that it's no longer hidden.

```
12  int main() {
13      cout << "global x in main is " << x << endl;
14
15      const int x{5}; // local variable to main
16
17      cout << "local x in main's outer scope is "
<< x << endl;
18
19      { // block starts a new scope
20          const int x{7}; // hides both x in outer
scope and global x
21
22          cout << "local x in main's inner scope is "
" << x << endl;
23      }
24
25      cout << "local x in main's outer scope is "
<< x << endl;
26
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5
```

To demonstrate other scopes, the program defines three functions—`useLocal`, `useStaticLocal` and `useGlobal`—each of which takes no arguments and returns nothing. The rest of `main` (shown below) calls each function twice in lines 27–32. After executing functions `useLocal`, `useStaticLocal` and `useGlobal` twice each, the program prints the local variable `x` in `main` again to show that none of the function calls modified the value of `x` in `main`, because the functions all referred to variables in other scopes.

```
27      useLocal(); // useLocal has local x
28      useStaticLocal(); // useStaticLocal has static
local x
29      useGlobal(); // useGlobal uses global x
```

```
30     useLocal(); // useLocal reinitializes its
local x
31     useStaticLocal(); // static local x retains
its prior value
32     useGlobal(); // global x also retains its
prior value
33
34     cout << "\nlocal x in main is " << x << endl;
35 }
36
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

## Function `useLocal`

Function `useLocal` initializes local variable `x` to 25 (line 39). When `main` calls `useLocal` (lines 27 and 30), the function prints the variable `x`, increments it and prints it again before the function returns program control to its caller. Each time the program calls this function, the function recreates local variable `x` and reinitializes it to 25.

```
37 // useLocal reinitializes local variable x during
each call
38 void useLocal() {
39     int x{25}; // initialized each time useLocal
is called
40
41     cout << "\nlocal x is " << x << " on entering
useLocal" << endl;
42     ++x;
```

```
43     cout << "local x is " << x << " on exiting  
useLocal" << endl;  
44 }  
45
```

---

### Function `useStaticLocal`

Function `useStaticLocal` declares `static` variable `x` and initializes it to `50`. Local variables declared as `static` retain their values even when they're out of scope (i.e., the function in which they're declared is not executing). When line 28 in `main` calls `useStaticLocal`, the function prints its local `x`, increments it and prints it again before the function returns program control to its caller. In the next call to this function (line 31), `static` local variable `x` contains the value `51`. The initialization in line 50 occurs only the first time `useStaticLocal` is called (line 28).

```
46 // useStaticLocal initializes static local  
variable x only the  
47 // first time the function is called; value of x  
is saved  
48 // between calls to this function  
49 void useStaticLocal() {  
50     static int x{50}; // initialized first time  
useStaticLocal is called  
51  
52     cout << "\nlocal static x is " << x << " on  
entering useStaticLocal"  
53         << endl;  
54     ++x;  
55     cout << "local static x is " << x << " on  
exiting useStaticLocal"  
56         << endl;  
57 }  
58
```

---

### Function `useGlobal`

Function `useGlobal` does not declare any variables. Therefore, when it refers to variable `x`, the global `x` (line 10, preceding `main`) is used. When `main` calls `useGlobal` (line 29), the function prints the global variable `x`, multiplies it by 10 and prints it again before the function returns to its caller. The next time `main` calls `useGlobal` (line 32), the global variable has its modified value, `10`.

---

```
59 // useGlobal modifies global variable x during
60 void useGlobal() {
61     cout << "\nglobal x is " << x << " on entering
useGlobal" << endl;
62     x *= 10;
63     cout << "global x is " << x << " on exiting
useGlobal" << endl;
64 }
```

---

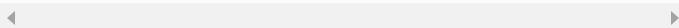
## 5.12 INLINE FUNCTIONS

Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead. C++ provides **inline functions** to help reduce function-call overhead. Placing **inline** before a function's return type in the function definition advises the compiler to generate a copy of the function's body code in every place where the function is called (when appropriate) to avoid a function call. This often makes the program larger. The compiler can ignore the **inline** qualifier. Reusable **inline** functions are typically placed in headers so that their definitions can be inlined in each source file that uses them.

If you change an **inline** function's definition, you must recompile any code that calls that function. Though compilers can inline code for which you have not explicitly used **inline**, the ISO's C++ Core Guidelines indicate that you should declare “small and time-critical” functions **inline**.<sup>14</sup> ISO also provides an extensive FAQ on the subtleties of using **inline** functions:

---

<https://isocpp.org/wiki/faq/inline-functions>



14. C++ Core Guidelines. Accessed May 11, 2020.

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline>.

Figure 5.8 uses **inline** function `cube` (lines 9–11) to calculate the volume of a cube.)

---

```
1 // fig05_08.cpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using namespace std;
5
6 // Definition of inline function cube. Definition of function appears
7 // before function is called, so a function prototype is not required.
8 // First line of function definition also acts as the prototype.
9 inline double cube(double side) {
10     return side * side * side; // calculate cube
11 }
12
13 int main() {
14     double sideValue; // stores value entered by user
15     cout << "Enter the side length of your cube: ";
16     cin >> sideValue; // read value from user
17
18     // calculate cube of sideValue and display result
19     cout << "Volume of cube with side "
20         << sideValue << " is " << cube(sideValue) << endl;
21 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

**Fig. 5.8** inline function that calculates the volume of a cube.

## 5.13 REFERENCES AND REFERENCE PARAMETERS

 **Perf** Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed by value, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. So far, each argument in the book has been passed by

value. One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

## Reference Parameters

This section introduces reference parameters—the first of the two means C++ provides for performing pass-by-reference.<sup>15</sup> When a variable is passed by reference, the caller gives the called function the ability to access that variable in the caller directly and to modify the variable.

<sup>15</sup>. Chapter 7 discusses pointers, which enable an alternate form of pass-by-reference in which the style of the function call clearly indicates pass-by-reference (and the potential for modifying the caller's arguments).



**Perf** Pass-by-reference is good for performance reasons because it can eliminate the pass-by-value overhead of copying large amounts of data. But Pass-by-reference can weaken security—the called function can corrupt the caller's data.



**Security** After this section's example, we'll show how to achieve the performance advantage of pass-by-reference while simultaneously achieving the software engineering advantage of protecting the caller's data from corruption.

A **reference parameter** is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header. For example, the parameter declaration

```
int& number
```

when reading from right to left is pronounced “number is a reference to an int.” As always, the function prototype and header must agree.

In the function call, simply mention a variable by name to pass it by reference. In the called function's body, the reference

parameter (e.g., `number`) refers to the original variable in the caller, which can be modified directly by the called function.

### Passing Arguments by Value and by Reference

Figure 5.9 compares pass-by-value and pass-by-reference with reference parameters. The “styles” of the arguments in the calls to function `squareByValue` and function `squareByReference` are identical—both variables are simply mentioned by name in the function calls. The compiler checks the function prototypes and definitions to determine whether to use pass-by-value or pass-by-reference.

---

```
1 // fig05_09.cpp
2 // Passing arguments by value and by reference.
3 #include <iostream>
4 using namespace std;
5
6 int squareByValue(int x); // function prototype (for value pass)
7 void squareByReference(int& z); // function prototype (for reference pass)
8
9 int main() {
10    const int x{2}; // value to square using squareByValue
11    int z{4}; // value to square using squareByReference
12
13    // demonstrate squareByValue
14    cout << "x = " << x << " before squareByValue\n";
```

```

15 cout << "Value returned by squareByValue: "
16     << squareByValue(x) << endl;
17 cout << "x = " << x << " after squareByValue\n" << endl;
18
19 // demonstrate squareByReference
20 cout << "z = " << z << " before squareByReference" << endl;
21 squareByReference(z);
22 cout << "z = " << z << " after squareByReference" << endl;
23 }
24
25 // squareByValue multiplies number by itself, stores the
26 // result in number and returns the new value of number
27 int squareByValue(int number) {
28     return number *= number; // caller's argument not modified
29 }
30
31 // squareByReference multiplies numberRef by itself and stores the result
32 // in the variable to which numberRef refers in function main
33 void squareByReference(int& numberRef) {
34     numberRef *= numberRef; // caller's argument modified
35 }

```

```

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference

```

**Fig. 5.9** Passing arguments by value and by reference.  
(Part 1 of 2.)

### References as Aliases within a Function

References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 5.9). For example, the code

---

```

int count{1}; // declare integer variable count
int& cRef{count}; // create cRef as an alias for
                  // increment count (using its alias cRef

```



increments variable `count` by using its alias `cRef`. Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables. In this sense, references are constant. All operations performed on the alias (i.e., the reference) are actually performed on the original variable. The alias is simply another name for the original variable. Unless it's a reference to a constant (discussed below), a reference's initializer must be an *lvalue*—something that can appear on the left side of an assignment, like a variable name. A reference may not be initialized with a constant or *rvalue* expression—that is, something might appear on the right side of an assignment, such as the result of a calculation.

### const References

To specify that a reference parameter should not be allowed to modify the corresponding argument in the caller, place the **const** qualifier before the type name in the parameter's declaration. For example, consider a `displayName` function:

```
void displayName(std::string name) {  
    std::cout << name << std::endl;  
}
```

When called, it receives a copy of its `string` argument. Since `string` objects can be large, this copy operation could degrade an application's performance. For this reason, `string` objects (and objects in general) should be passed to functions by reference.

Also, the `displayName` function does not need to modify its argument, so following the principle of least privilege, we'd declare the parameter as

```
const std::string& name
```

  **Perf** **Security** Reading this from right-to-left, the `name` parameter is a reference to a `string` constant. We get the

performance of passing the `string` by reference. Also, `displayName` treats the argument as a constant, so `displayName` cannot modify the value in the caller—so we get the security of pass-by-value.

### Returning a Reference to a Local Variable

Functions can return references to local variables, but this can be dangerous. When returning a reference to a local non-`static` variable, the reference refers to a variable that's discarded when the function terminates. An attempt to access such a variable yields undefined behavior, often crashing the program or corrupting data.<sup>16</sup> References to undefined variables are called **dangling references**. This is a logic error for which compilers typically issue a warning. Software-engineering teams often have policies requiring that before code can be deployed, it must compile without warnings.

16. C++ Core Guidelines. Accessed May 11, 2020.  
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-dangle>.

## 5.14 DEFAULT ARGUMENTS

It's common for a program to invoke a function repeatedly with the same argument value for a particular parameter. In such cases, you can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter. When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call, inserting the default value of that argument.

### boxVolume Function with Default Arguments

Figure 5.10 demonstrates using default arguments to calculate a box's volume. The function prototype for `boxVolume` (line 7) specifies that all three parameters have default values of `1` by placing `= 1` to the right of each parameter.

---

1 // fig05\_10.cpp  
2 // Using default arguments.

```

3 #include <iostream>
4 using namespace std;
5
6 // function prototype that specifies default arguments
7 int boxVolume(int length = 1, int width = 1, int height = 1);
8
9 int main() {
10    // no arguments--use default values for all dimensions
11    cout << "The default box volume is: " << boxVolume();
12
13    // specify length; default width and height
14    cout << "\n\nThe volume of a box with length 10,\n"
15        << "width 1 and height 1 is: " << boxVolume(10);
16
17    // specify length and width; default height
18    cout << "\n\nThe volume of a box with length 10,\n"
19        << "width 5 and height 1 is: " << boxVolume(10, 5);
20
21    // specify all arguments
22    cout << "\n\nThe volume of a box with length 10,\n"
23        << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
24        << endl;
25 }
26
27 // function boxVolume calculates the volume of a box
28 int boxVolume(int length, int width, int height) {
29    return length * width * height;
30 }

```

The default box volume is: 1

The volume of a box with length 10,  
width 1 and height 1 is: 10

The volume of a box with length 10,  
width 5 and height 1 is: 50

The volume of a box with length 10,  
width 5 and height 2 is: 100

**Fig. 5.10** Using default arguments. (Part 1 of 2.)

The first call to `boxVolume` (line 11) specifies no arguments, thus using all three default values of 1. The second call (line 15) passes only a `length` argument, thus using default values of 1 for the `width` and `height` arguments. The third call (line 19) passes arguments for only `length` and `width`, thus using a default value of 1 for the `height` argument. The last call (line 23) passes arguments for `length`, `width` and `height`, thus using no default values. Any arguments passed to the function explicitly are assigned to the function's parameters from left to right. Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list). When `boxVolume` receives two arguments, the function assigns the values of those arguments to its `length` and `width` parameters in that order. Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to its `length`, `width` and `height` parameters, respectively.

### Notes Regarding Default Arguments

Default arguments must be the rightmost (trailing) arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument, then all arguments to the right of that argument also must be omitted. Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype. If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header. Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used with `inline` functions. Using default arguments can simplify writing function calls, but some programmers feel that explicitly specifying all arguments is clearer.

## 5.15 UNARY SCOPE RESOLUTION OPERATOR

C++ provides the **unary scope resolution operator (::)** to access a global variable when a local variable of the same name is in scope. The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Figure 5.11 shows the unary scope resolution operator with local and global variables of the same name (lines 6 and 9). To emphasize that the local and global versions of variable `number` are distinct, the program declares one variable `int` and the other `double`.

```
1 // fig05_11.cpp
2 // Unary scope resolution operator.
3 #include <iostream>
4 using namespace std;
5
6 int number{7}; // global variable named number
7
8 int main() {
9     double number{10.5}; // local variable named number
10
11    // display values of local and global variables
12    cout << "Local double value of number = " << number
13    << "\nGlobal int value of number = " << ::number << endl;
14 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

**Fig. 5.11** Unary scope resolution operator.

Always use the unary scope resolution operator (::) to refer to global variables (even if there is no collision with a local-variable name). This makes it clear that you're accessing a global variable rather than a local variable. It also makes programs easier to modify by reducing the risk of name

collisions with nonglobal variables and eliminates logic errors that might occur if a local variable hides the global variable.

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.

## 5.16 FUNCTION OVERLOADING

C++ enables several functions of the same name to be defined, as long as they have different signatures. This is called

**function overloading**. The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call. Function overloading is used to create several functions of the same name that perform similar tasks but on data of different types. For example, many functions in the math library are overloaded for different numeric types—the C++ standard requires `float`, `double` and `long double` overloaded versions of the math library functions in Section 5.3. Overloading functions that perform closely related tasks can make programs clearer.

### Overloaded `square` Functions

Figure 5.12 uses overloaded `square` functions to calculate the square of an `int` (lines 7–10) and the square of a `double` (lines 13–16). Line 19 invokes the `int` version of function `square` by passing the literal value 7. C++ treats whole-number literal values as type `int`. Similarly, line 21 invokes the `double` version of function `square` by passing the literal value `7.5`, which C++ treats as a `double`. In each case, the compiler chooses the proper function to call, based on the type of the argument. The output confirms that the proper function was called in each case.

---

```
1 // fig05_12.cpp
2 // Overloaded square functions.
3 #include <iostream>
4 using namespace std;
5
6 // function square for int values
7 int square(int x) {
8     cout << "square of integer " << x << " is ";
9     return x * x;
10 }
11
12 // function square for double values
13 double square(double y) {
14     cout << "square of double " << y << " is ";
15     return y * y;
16 }
17
18 int main() {
19     cout << square(7); // calls int version
20     cout << endl;
21     cout << square(7.5); // calls double version
22     cout << endl;
23 }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

**Fig. 5.12** Overloaded square functions. (Part 1 of 2.)

## How the Compiler Differentiates Among Overloaded Functions

Overloaded functions are distinguished by their signatures. A signature is a combination of a function's name and its parameter types (in order). **Type-safe linkage** ensures that the proper function is called and that the types of the arguments conform to the types of the parameters. To enable type-safe linkage, the compiler internally encodes each function identifier with the types of its parameters—a process referred to as **name mangling**. These encodings vary by compiler, so

everything that will be linked to create an executable for a given platform must be compiled using the same compiler for that platform. Figure 5.13 was compiled with GNU C++.<sup>17</sup>

Rather than showing the execution output of the program as we normally would, we show the mangled function names produced in assembly language by GNU C++.<sup>18</sup>

---

<sup>17</sup>. The empty-bodied `main` function ensures that we do not get a linker error if we compile this code.

<sup>18</sup>. The command `g++ -S fig05_13.cpp` produces the assembly language file `fig05_14.s`.

---

```
1 // fig05_13.cpp
2 // Name mangling to enable type-safe linkage.
3
4 // function square for int values
5 int square(int x) {
6     return x * x;
7 }
8
9 // function square for double values
10 double square(double y) {
11     return y * y;
12 }
13
14 // function that receives arguments of types
15 // int, float, char and int&
16 void nothing1(int a, float b, char c, int& d) { }
17
18 // function that receives arguments of types
19 // char, int, float& and double&
20 int nothing2(char a, int b, float& c, double& d) {
21     return 0;
22 }
23
24 int main() { }
```

```
_Z6squarei  
_Z6squared  
_Z8nothing1ifcRi  
_Z8nothing2ciRfRd  
main
```

**Fig. 5.13** Name mangling to enable type-safe linkage.  
(Part 1 of 2.)

For GNU C++, each mangled name (other than `main`) begins with an underscore (`_`) followed by the letter Z, a number and the function name. The number that follows Z specifies how many characters are in the function's name. For example, function `square` has 6 characters in its name, so its mangled name is prefixed with `_Z6`. Following the function name is an encoding of its parameter list:

- For function `square` that receives an `int` (line 5), i represents `int`, as shown in the output's first line.
- For function `square` that receives a `double` (line 10), d represents `double`, as shown in the output's second line.
- For function `nothing1` (line 16), i represents an `int`, f represents a `float`, c represents a `char`, and Ri represents an `int&` (i.e., a reference to an `int`), as shown in the output's third line.
- For function `nothing2` (line 20), c represents a `char`, i represents an `int`, Rf represents a `float&`, and Rd represents a `double&`.

The compiler distinguishes the two `square` functions by their parameter lists—one specifies i for `int` and the other d for `double`. The return types of the functions are not specified in the mangled names. Overloaded functions can have different return types, but if they do, they must also have different parameter lists. Function-name mangling is compiler-specific. For example, Visual C++ produces the name

`square@@YAH@Z` for the square function at line 5. The GNU C++ compiler did not mangle `main`'s name, but some compilers do. For example, Visual C++ uses `_main`.

Creating overloaded functions with identical parameter lists and different return types is a compilation error. The compiler uses only the parameter lists to distinguish between overloaded functions. Such functions need not have the same number of parameters.

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler cannot unambiguously determine which version of the function to choose.

## Overloaded Operators

In Chapter 14, we discuss how to overload operators to define how they should operate on objects of user-defined data types. (In fact, we've been using many overloaded operators to this point, including the stream insertion `<<` and the stream extraction `>>` operators. These are overloaded for all the fundamental types. We say more about overloading `<<` and `>>` to be able to handle objects of user-defined types in Chapter 14.)

## 5.17 FUNCTION TEMPLATES

Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using **function templates**. You write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate **function template specializations** to

handle each type of call appropriately. Thus, defining a single function template essentially defines a whole family of overloaded functions.

### maximum Function Template

Figure 5.14 defines a `maximum` function template that determines the largest of three values. All function template definitions begin with the **template keyword** (line 3) followed by a **template parameter list** enclosed in angle brackets (< and >). Every parameter in the template parameter list is preceded by keyword **typename** or keyword **class** (they are synonyms in this context). The **type parameters** are placeholders for fundamental types or user-defined types. These placeholders—in this case, `T`—are used to specify the types of the function’s parameters (line 4), to specify the function’s return type (line 4) and to declare variables within the body of the function definition (line 5). A function template is defined like any other function but uses the type parameters as placeholders for actual data types.

---

```
1 // Fig. 5.14: maximum.h
2 // Function template maximum header.
3 template <typename T> // or template<class T>
4 T maximum(T value1, T value2, T value3) {
5     T maximumValue{value1}; // assume value1 is maximum
6
7     // determine whether value2 is greater than maximumValue
8     if (value2 > maximumValue) {
9         maximumValue = value2;
10    }
11
12    // determine whether value3 is greater than maximumValue
13    if (value3 > maximumValue) {
14        maximumValue = value3;
15    }
16
17    return maximumValue;
18 }
```

---

**Fig. 5.14** Function template `maximum` header.

This function template declares a single type parameter `T` (line 3) as a placeholder for the type of the data to be tested by function `maximum`. The name of a type parameter must be unique in the template parameter list for a particular template definition. When the compiler encounters a call to `maximum` in the program source code, the compiler substitutes the argument types in the `maximum` call for `T` throughout the template definition, creating a complete function template specialization that determines the maximum of three values of the specified type. The values must have the same type, since we use only one type parameter in this example. Then the newly created function is compiled—templates are a means of code generation. We'll use C++ Standard Library templates that require multiple type parameters in Chapter 16.

### Using Function Template `maximum`

Figure 5.15 uses the `maximum` function template to determine the largest of three `int` values, three `double` values and three `char` values, respectively (lines 15, 24 and 33). Because each call uses arguments of a different type, “behind the scenes” the compiler creates a separate function definition for each—one expecting three `int` values, one expecting three `double` values and one expecting three `char` values, respectively.

---

```
1 // fig05_15.cpp
2 // Function template maximum test program.
3 #include <iostream>
4 #include "maximum.h" // include definition of function template maximum
5 using namespace std;
6
7 int main() {
8     // demonstrate maximum with int values
9     cout << "Input three integer values: ";
10    int int1, int2, int3;
11    cin >> int1 >> int2 >> int3;
12
13    // invoke int version of maximum
14    cout << "The maximum integer value is: "
15        << maximum(int1, int2, int3);
16
17    // demonstrate maximum with double values
18    cout << "\n\nInput three double values: ";
19    double double1, double2, double3;
20    cin >> double1 >> double2 >> double3;
21
22    // invoke double version of maximum
23    cout << "The maximum double value is: "
24        << maximum(double1, double2, double3);
25
26    // demonstrate maximum with char values
27    cout << "\n\nInput three characters: ";
28    char char1, char2, char3;
29    cin >> char1 >> char2 >> char3;
30
31    // invoke char version of maximum
32    cout << "The maximum character value is: "
33        << maximum(char1, char2, char3) << endl;
34 }
```

```
Input three integer values: 1 2 3  
The maximum integer value is: 3
```

```
Input three double values: 3.3 2.2 1.1  
The maximum double value is: 3.3
```

```
Input three characters: A C B  
The maximum character value is: C
```

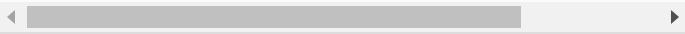
**Fig. 5.15** Function template `maximum` test program.

(Part 1 of 2.)

### maximum Function Template Specialization for Type int

The function template specialization created for type `int` replaces each occurrence of `T` with `int` as follows:

```
int maximum(int value1, int value2, int value3)  
    int maximumValue{value1}; // assume value1 is  
  
    // determine whether value2 is greater than m  
    if (value2 > maximumValue) {  
        maximumValue = value2;  
    }  
  
    // determine whether value3 is greater than m  
    if (value3 > maximumValue) {  
        maximumValue = value3;  
    }  
  
    return maximumValue;  
}
```



## 5.18 RECURSION

For some problems, it's useful to have functions call themselves. A **recursive function** is a function that calls itself, either directly or indirectly (through another function). This section and the next present simple examples of recursion. Recursion is discussed at length in upper-level computer science courses.

### Recursion Concepts

We first consider recursion conceptually, then examine programs containing recursive functions. Recursive problem-solving approaches have several elements in common. A recursive function is called to solve a problem. The function knows how to solve only the simplest case(s), or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step often includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.

Omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case causes an infinite recursion error, typically causing a stack overflow. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Accidentally having a nonrecursive function call itself, either directly or indirectly through another function, also causes an infinite recursion.

The recursion step executes while the original call to the function is still “open,” i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces. For the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function. Then, a sequence of returns ensues up the line until the original call eventually

returns the final result to `main`.<sup>19</sup> This sounds quite exotic compared to the kind of problem-solving we've been using to this point. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

<sup>19</sup>. The C++ standard document indicates that `main` should not be called within a program (Section 6.9.3.1) or recursively (Section 7.6.1.2). Its sole purpose is to be the starting point for program execution.

## Factorial

The factorial of a non-negative integer  $n$ , written  $n!$  (pronounced “ $n$  factorial”), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

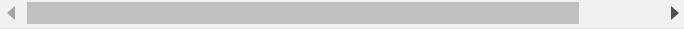
with  $1!$  equal to 1, and  $0!$  defined to be 1. For example,  $5!$  is the product  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , which is equal to 120.

## Iterative Factorial

The factorial of an integer, `number`, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) by using a `for` statement as follows:

```
int factorial[1];

for (int counter{number}; counter >= 1; --counter)
    factorial *= counter;
}
```



## Recursive Factorial

A recursive definition of the factorial function is arrived at by observing the following algebraic relationship:

$$n! = n \cdot (n - 1)!$$

For example,  $5!$  is clearly equal to  $5 * 4!$  as is shown by the following:

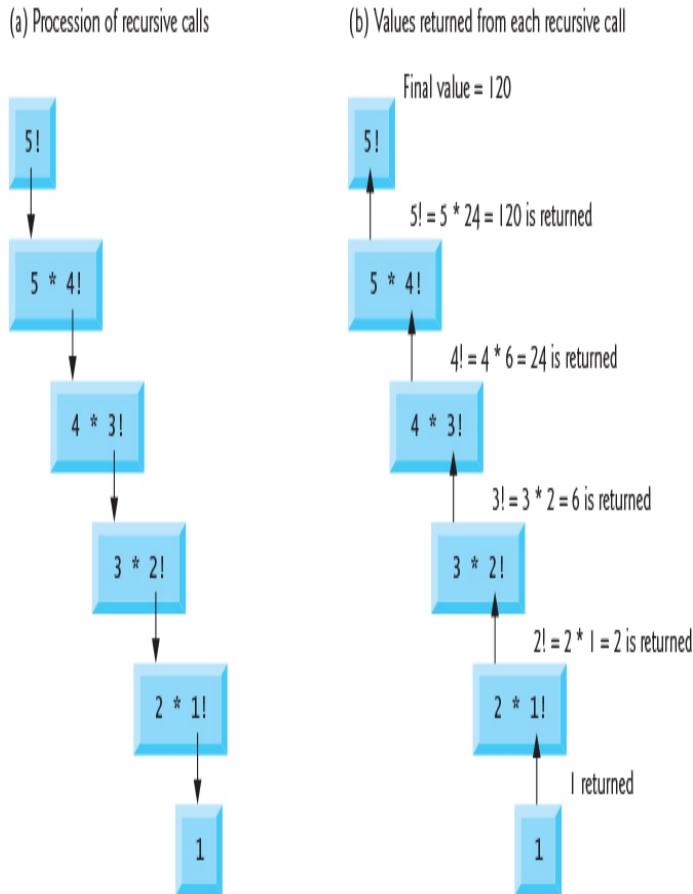
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

## Evaluating 5!

The evaluation of  $5!$  would proceed as shown in the following diagram, which illustrates how the succession of recursive calls proceeds until  $1!$  is evaluated to be 1, terminating the recursion. Part (b) of the diagram shows the values returned from each recursive call to its caller until the final value is calculated and returned.



### Using a Recursive `factorial` Function to Calculate Factorials

Figure 5.16 uses recursion to calculate and print the factorials of the integers 0–10. The recursive function `factorial` (lines 18–25) first determines whether the terminating condition `number <= 1` (i.e., the base case; line 19) is true. If `number` is less than or equal to 1, the `factorial` function returns 1 (line 20), no further recursion is necessary and the function terminates. If `number` is greater than 1, line 23 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of

`number - 1`, which is a slightly simpler problem than the original calculation `factorial(number)`.

---

```
1 // fig05_16.cpp
2 // Recursive function factorial.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 long factorial(long number); // function prototype
8
9 int main() {
10    // calculate the factorials of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12        cout << setw(2) << counter << "!" = " << factorial(counter)
```

```

13     << endl;
14 }
15 }
16
17 // recursive definition of function factorial
18 long factorial(long number) {
19     if (number <= 1) { // test for base case
20         return 1; // base cases: 0! = 1 and 1! = 1
21     }
22     else { // recursion step
23         return number * factorial(number - 1);
24     }
25 }
```

$0! = 1$   
 $1! = 1$   
 $2! = 2$   
 $3! = 6$   
 $4! = 24$   
 $5! = 120$   
 $6! = 720$   
 $7! = 5040$   
 $8! = 40320$   
 $9! = 362880$   
 $10! = 3628800$

**Fig. 5.16** Recursive function `factorial`. (Part 1 of 2.)

### Factorial Values Grow Quickly

Function `factorial` receives a parameter of type `long` and returns a result of type `long`. Typically, a `long` is stored in at least four bytes (32 bits); such a variable can hold a value in the range  $-2,147,483,647$  to  $2,147,483,647$ . Unfortunately, the function `factorial` produces large values so quickly that type `long` does not help us compute many factorial values before reaching the maximum value of a `long`. For larger integer values, we could use type `long long` (Section 3.11) or a class that represents arbitrary sized integers (such as the

open-source `BigNumber` class we introduced in Section 3.12).

## 5.19 EXAMPLE USING RECURSION: FIBONACCI SERIES

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of a spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618.... This number frequently occurs in nature and has been called the **golden ratio** or the **golden mean**. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio. A web search for “Fibonacci in nature” reveals many interesting examples, including flower petals, shells, spiral galaxies, hurricanes and more.

### Recursive Fibonacci Definition

The Fibonacci series can be defined recursively as follows:

$$\begin{aligned}\text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)\end{aligned}$$

The program of Fig. 5.17 calculates the  $n$ th Fibonacci number recursively by using function `fibonacci`. Fibonacci numbers tend to become large quickly, although slower than factorials do. Figure 5.17 shows the execution of the program, which displays the Fibonacci values for several numbers.

---

```
1 // fig05_17.cpp
2 // Recursive function fibonacci.
3 #include <iostream>
4 using namespace std;
5
6 long fibonacci(long number); // function prototype
7
8 int main() {
9     // calculate the fibonacci values of 0 through 10
10    for (int counter{0}; counter <= 10; ++counter)
11        cout << "fibonacci(" << counter << ")" = "
12            << fibonacci(counter) << endl;
13
14    // display higher fibonacci values
15    cout << "\nfibonacci(20) = " << fibonacci(20) << endl;
16    cout << "fibonacci(30) = " << fibonacci(30) << endl;
17    cout << "fibonacci(35) = " << fibonacci(35) << endl;
18 }
19
20 // recursive function fibonacci
21 long fibonacci(long number) {
22     if ((0 == number) || (1 == number)) { // base cases
23         return number;
24     }
25     else { // recursion step
26         return fibonacci(number - 1) + fibonacci(number - 2);
27     }
28 }
```

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55

fibonacci(20) = 6765
fibonacci(30) = 832040
fibonacci(35) = 9227465
```

**Fig. 5.17** Recursive function `fibonacci`. (Part 1 of 2.)

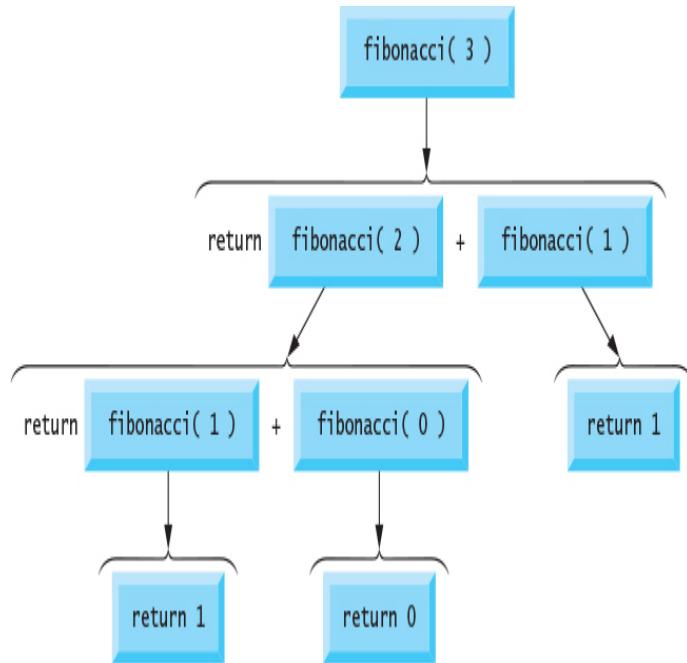
The application begins with a loop that calculates and displays the Fibonacci values for the integers 0–10 and is followed by three calls to calculate the Fibonacci values of the integers 20, 30 and 35 (lines 15–17). The calls to `fibonacci` in `main` (lines 12 and 15–17) are not recursive calls, but the calls from line 26 of `fibonacci` are recursive. Each time the program invokes `fibonacci` (lines 21–28), the function immediately tests the base case to determine whether `number` is equal to 0 or 1 (line 22). If this is true, line 23 returns `number`.

Interestingly, if `number` is greater than 1, the recursion step (line 26) generates two recursive calls, each for a slightly smaller problem than the original call to `fibonacci`.

### Evaluating `fibonacci(3)`

The following diagram shows how function `fibonacci` would evaluate `fibonacci(3)`. This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators. This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and grouping. The following diagram shows that

evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`. In what order are these calls made?



### Order of Evaluation of Operands

Most programmers simply assume that the operands are evaluated left to right, which is the case in some programming languages. C++ does not specify the order in which the operands of many operators (including `+`) are to be evaluated. Therefore, you must make no assumption about the order in which these calls execute. The calls could, in fact, execute `fibonacci(2)` first, then `fibonacci(1)`, or `fibonacci(1)` first, then `fibonacci(2)`. In this program and in most others, it turns out that the final result would be the same. However, in some programs, the evaluation of an operand can have **side effects** (changes to data values) that could affect the final result of the expression.

### Operators for which Order Of Evaluation Is Specified

Before C++17, C++ specified the order of evaluation of the operands of only the operators `&&`, `||`, comma `(, )` and `?:`. The first three are binary operators whose two operands are guaranteed to be evaluated left to right. The last operator is

C++’s only ternary operator— its leftmost operand is always evaluated first; if it evaluates to true, the middle operand evaluates next and the last operand is ignored; if the leftmost operand evaluates to false, the third operand evaluates next and the middle operand is ignored.

**17** As of C++17, C++ now also specifies the order of evaluation of the operands for various other operators. For the operators dot ( . ), [ ] ([Chapter 6](#)), -> ([Chapter 7](#)), parentheses (of a function call), <<, >> and ->\*, the compiler evaluates the operands left-to-right. For a function call’s parentheses, this means that the compiler evaluates the function name before the arguments. The compiler evaluates the operands of assignment operators right-to-left.

Writing programs that depend on the order of evaluation of the operands of other operators can lead to logic errors. For other operators, to ensure that side effects are applied in the correct order, break complex expressions into separate statements.

Recall that the `&&` and `||` operators use short-circuit evaluation. Placing an expression with a side effect on the right side of the `&&` or `||` operator is a logic error if that expression should always be evaluated.

## Exponential Complexity

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in function `fibonacci` has a doubling effect on the number of function calls; i.e., the number of recursive calls that are required to calculate the  $n$ th Fibonacci number is on the order of  $2^n$ . This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of  $2^{20}$  or about a million calls, calculating the 30th Fibonacci number would require on the order of  $2^{30}$  or about a billion calls, and so on. Computer scientists refer to this as **exponential complexity**.

Problems of this nature can humble even the world’s most powerful computers as  $n$  becomes large.

Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science course typically called Algorithms. Avoid Fibonacci-

style recursive programs that result in an exponential “explosion” of calls.

## 5.20 RECURSION VS. ITERATION

In the two prior sections, we studied two recursive functions that can also be implemented with simple iterative programs. This section compares the two approaches and discusses why you might choose one approach over the other in a particular situation.

- Both iteration and recursion are based on a control statement: Iteration uses an iteration statement; recursion uses a selection statement.
- Both iteration and recursion involve iteration: Iteration explicitly uses an iteration statement; recursion achieves iteration through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Counter-controlled iteration and recursion each gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion produces simpler versions of the original problem until the base case is reached.
- Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

### Iterative Factorial Implementation

To illustrate the differences between iteration and recursion, let’s examine an iterative solution to the factorial problem (Fig. 5.18). Lines 22–24 use an iteration statement rather than the selection statement of the recursive solution (lines 19–24

of Fig. 5.16). Both solutions use a termination test. In the recursive solution, line 19 (Fig. 5.16) tests for the base case. In the iterative solution, line 22 (Fig. 5.18) tests the loop-continuation condition—if the test fails, the loop terminates. Finally, instead of producing simpler versions of the original problem, the iterative solution uses a counter that's modified until the loop-continuation condition becomes false.

---

```
1 // fig05_18.cpp
2 // Iterative function factorial.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 unsigned long factorial(int number); // function prototype
8
9 int main() {
10    // calculate the factorials of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12        cout << setw(2) << counter << "!" = " << factorial(counter)
13        << endl;
14    }
15 }
16
17 // iterative function factorial
18 unsigned long factorial(int number) {
19    unsigned long result{1};
20
21    // iterative factorial calculation
22    for (int i{number}; i >= 1; --i) {
23        result *= i;
24    }
25
26    return result;
27 }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

**Fig. 5.18** Iterative function factorial.

## Negatives of Recursion

Recursion has negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function variables to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

## When to Choose Recursion vs. Iteration

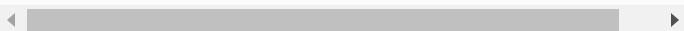


Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent when a recursive solution is. If possible, avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

## 17 5.21 C++17 AND C++20: [[nodiscard]] ATTRIBUTE

Some functions return values that you should not ignore. For example, Section 2.7 introduced the `string` member function `empty`. When you want to know whether a `string` is empty you must not only call `empty`, but also check its return value in a condition, such as:

```
if (s.empty()) {  
    // do something because the string s is empty  
}
```



As of C++17, `string`'s `empty` function is declared with the **[[nodiscard]] attribute**<sup>20</sup> to tell the compiler to issue a warning when the return value is not used by the caller. Since

C++17, many C++ Standard Library functions have been enhanced with `[[nodiscard]]` so the compiler can help you write correct code.

20. Section 9.12.8 of the ISO/IEC C++20 standard document.

<https://wg21.link/n4849>.

You may also use this attribute on your own function definitions. [Figure 5.19](#) shows a `cube` function declared with `[[nodiscard]]`, which you place before the return type—typically on a line by itself (line 4).

---

```
1 // fig05_19.cpp
2 // C++17 [[nodiscard]] attribute.
3
4 [[nodiscard]]
5 int cube(int x) {
6     return x * x * x;
7 }
8
9 int main() {
10    cube(10); // generates a compiler warning
11 }
```

---

**Fig. 5.19** C++17 `[[nodiscard]]` attribute.

Line 10 calls `cube`, but does not use the returned value. When you compile this program our preferred compilers, they issue the following warnings:

- Microsoft Visual C++: "discarding return value of function with 'nodiscard' attribute"
- Clang in Xcode: "Ignoring return value of function declared with 'nodiscard' attribute"
- GNU C++: "ignoring return value of 'int cube(int)', declared with attribute `nodiscard`"

However, these are just warnings, so the program still compiles and runs.

## 20 C++20's `[[nodiscard("with reason")]]` Attribute

One problem with C++17's `[[nodiscard]]` attribute is that it did not provide any insight into why you should not ignore a given function's return value. So, in C++20, you can now include a message<sup>21</sup> that will be displayed as part of the compiler warning, as in:

```
[[nodiscard("Insight into why return value shou
```



<sup>21</sup>. At the time of this writing, this feature is not yet implemented.

## 5.22 LNFYLUN LHQTOMH WJTZ QARCV: QJWAZRPLM XZZ XNDMWWQHLZ

No doubt, you've noticed that the last Objectives bullet for this chapter, the last section name in the chapter outline, the last sentence in Section 5.1 and the section title above all look like gibberish. These are not mistakes! In this section, we continue our objects-natural presentation. You'll conveniently encrypt and decrypt messages with an object you create of a preexisting class that implements a **Vignère secret key cipher**.<sup>22</sup>

<sup>22</sup>. [https://en.wikipedia.org/wiki/Vigen%C3%A8re\\_cipher](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher).

In prior objects-natural sections, you created objects of built-in C++ standard library class `string` and objects of classes from open-source libraries. Sometimes you'll use classes built by your organization or team members for internal use or for use in a specific project. For this example, we wrote our own class called `Cipher` (in the header "cipher.h") and provided it to you. In [Chapter 10, Introduction to Classes](#), you'll start building your own custom classes.

## Cryptography

 **Security** Cryptography has been in use for thousands of years<sup>23,24</sup> and is critically important in today's connected

world. Every day, cryptography is used behind the scenes to ensure that your Internet-based communications are private and secure. For example, most websites now use the HTTPS protocol to encrypt and decrypt your web interactions.

23.

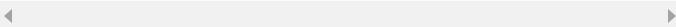
[https://en.wikipedia.org/wiki/Cryptography#History\\_of\\_cryptography\\_and\\_cryptanalysis](https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis).

24. <https://www.binance.vision/security/history-of-cryptography>.

## Caesar Cipher

Julius Caesar used a simple **substitution cipher** to encrypt military communications.<sup>25</sup> His technique—which became known as the **Caesar cipher**—replaces every letter in a communication with the letter three ahead in the alphabet. So, A is replaced with D, B with E, C with F, … X with A, Y with B and Z with C. Thus, the plain text

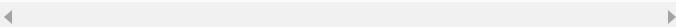
Caesar Cipher



25. [https://en.wikipedia.org/wiki/Caesar\\_cipher](https://en.wikipedia.org/wiki/Caesar_cipher).

would be encrypted as

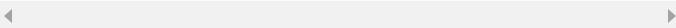
Fdhvdu Flskhu



The encrypted text is known as the **ciphertext**.

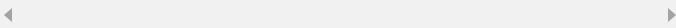
For a fun way to play with the Caesar cipher and many other cipher techniques, check out the website:

<https://cryptii.com/pipes/caesar-cipher>



which is an online implementation of the open-source **cryptii** project:

<https://github.com/cryptii/cryptii>



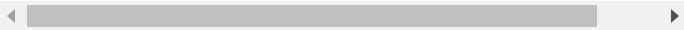
## Vignère Cipher

Simple substitution ciphers like the Caesar cipher are relatively easy to decrypt. For example, the letter “e” is the most frequently used letter in English. So, you could study ciphertext and assume with a high likelihood that the character appearing most frequently is probably an “e.”

In this example, you’ll use a Vignère cipher, which is a secret-key substitution cipher. A Vignère cipher is implemented using 26 Caesar ciphers—one for each letter of the alphabet. A Vignère cipher uses letters from the plain text and secret key to look up replacement characters in the various Caesar ciphers. You can read more about the implementation at

---

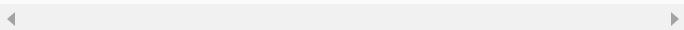
[https://en.wikipedia.org/wiki/Vigen%C3%A8re\\_cipher](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher)



For this cipher, the secret key must be composed of letters. Like passwords, the secret key should not be something that’s easy to guess. In this example, we used the 11 randomly selected characters

---

XMWUJBVYHXZ



There’s no limit to the number of characters you can use in your secret key. However, the person decrypting the ciphertext must know the secret key that was originally used to create the ciphertext.<sup>26</sup> Presumably, you’d provide that in advance—possibly in a face-to-face meeting.

<sup>26</sup>. There are many websites offering Vignère cipher decoders that attempt to decrypt ciphertext without the original secret key. We tried several, but none restored our original text.

## Using Our Cipher Class

For the example in Fig. 5.20, you’ll use our class `Cipher`, which implements the Vignère cipher. The header “`cipher.h`” (line 3) from this chapter’s `ch05` examples folder defines the class. You don’t need to read and understand the class’s code to use its encryption and decryption capabilities. As with all our other “Objects Natural” case studies, you can simply create an object of class `Cipher` and

call its `encrypt` and `decrypt` member functions to encrypt and decrypt text, respectively. In a later chapter, we'll present class `Cipher`'s implementation.

---

```
1 // fig15_20.cpp
2 // Encrypting and decrypting text with a Vigenère cipher.
3 #include "cipher.h"
4 #include <iostream>
5 #include <string>
```

```

6 using namespace std;
7
8 int main() {
9     string plainText;
10    cout << "Enter the text to encrypt:\n";
11    getline(cin, plainText);
12
13    string secretKey;
14    cout << "\nEnter the secret key:\n";
15    getline(cin, secretKey);
16
17    Cipher cipher;
18
19    // encrypt plainText using secretKey
20    string cipherText{cipher.encrypt(plainText, secretKey)};
21    cout << "\nEncrypted:\n " << cipherText << endl;
22
23    // decrypt cipherText
24    cout << "\nDecrypted:\n "
25    << cipher.decrypt(cipherText, secretKey) << endl;
26
27    // decrypt ciphertext entered by the user
28    cout << "\nEnter the ciphertext to decipher:\n";
29    getline(cin, cipherText);
30    cout << "\nDecrypted:\n "
31    << cipher.decrypt(cipherText, secretKey) << endl;
32 }

```

Enter the text to encrypt:  
**Welcome to Modern C++ application development with C++20!**

Enter the secret key:  
**XMWUJBVYHXZ**

Encrypted:  
**Tqhwxnz rv Jnaqnh L++ bknsfbxfeiw eztlinmyahc xdro Z++20!**

Decrypted:  
**Welcome to Modern C++ application development with C++20!**

Enter the ciphertext to decipher:  
**Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqh1z**

Decrypted:  
Objects Natural Case Study: Encryption and Decryption

**Fig. 5.20** Encrypting and decrypting text with a Vigenère cipher.

### Class `cipher`'s Member Functions

The class provides two key member functions:

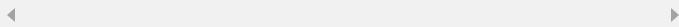
- Member function `encrypt` receives `strings` representing the plain text to encrypt and the secret key, uses the Vigenère cipher to encrypt the text, then returns a `string` containing the ciphertext.
- Member function `decrypt` receives `strings` representing the ciphertext to decrypt, reverses the Vigenère cipher to decrypt the text, then returns a `string` containing the plain text.

The program first asks you to enter text to encrypt and a secret key. Line 17 creates the `Cipher` object. Lines 20–21 encrypt the text you entered and display the encrypted text. Then, lines 24–25 decrypt the text to show you the plain-text string you entered.

Though the last Objectives bullet in this chapter, the last sentence of Section 5.1 and this section's title look like gibberish, they're each ciphertext that we created with our `Cipher` class and the secret key

---

XMWUJBVYHXZ



Line 28–29 prompt for and input existing ciphertext, then lines 30–31 decrypt the ciphertext and display the original plain text that we encrypted.

## 5.23 WRAP-UP

In this chapter, we presented function concepts, including function prototypes, function signatures, function headers and

function bodies. We overviewed the math library functions and new math functions and constants added in C++20, C++17 and C++11.

You learned about argument coercion—forcing arguments to the appropriate types specified by the parameter declarations of a function. We presented an overview of the C++ Standard Library’s headers. We demonstrated how to use functions `rand` and `srand` to generate random numbers that can be used for simulations, then presented C++11’s nondeterministic capabilities for producing more secure random numbers. We introduced C++14’s digit separators for more readable large numeric literals. We defined sets of constants with scoped `enums` and introduced C++20’s `using enum` declaration.

You learned about the scope of variables. We discussed two ways to pass arguments to functions—pass-by-value and pass-by-reference. We showed how to implement inline functions and functions that receive default arguments. You learned that overloaded functions have the same name but different signatures. Such functions can be used to perform the same or similar tasks, using different types or different numbers of parameters. We demonstrated using function templates to conveniently generate families of overloaded functions. You then studied recursion, where a function calls itself to solve a problem.

We presented C++17’s `[[nodiscard]]` attribute for indicating that a function’s return value should not be ignored and discussed C++20’s `[[nodiscard]]` enhancement for specifying a reason why the return value should not be ignored. Finally, our objects-natural case study introduced secret key substitution ciphers for encrypting and decrypting text.

In [Chapter 6](#), you’ll learn how to maintain lists and tables of data in arrays and object-oriented `vectors`. You’ll see a more elegant array-based implementation of the dice-rolling application.

## **Part 2: Arrays, Pointers, Strings and Files [This content is currently in development.]**

This content is currently in development.

## **6. Class Templates `array` and `vector`; Intro to C++20 Concepts and Ranges [This content is currently in development.]**

This content is currently in development.

## **7. Pointers [This content is currently in development.]**

**This content is currently in development.**

## 8. Class string and Regular Expressions [This content is currently in development.]

This content is currently in development.

## **9. File Processing and String Stream Processing [This content is currently in development.]**

**This content is currently in development.**

## **Part 3: Object-Oriented Programming [This content is currently in development.]**

This content is currently in development.

## **10. Introduction to Classes**

**[This content is currently in development.]**

**This content is currently in development.**

## **11. Classes and Objects: A Deeper Look [This content is currently in development.]**

**This content is currently in development.**

## **12. Inheritance [This content is currently in development.]**

**This content is currently in development.**

## **13. Polymorphism [This content is currently in development.]**

**This content is currently in development.**

## **14. Operator Overloading [This content is currently in development.]**

**This content is currently in development.**

## **15. Exceptions: A Deeper Look**

**[This content is currently in development.]**

**This content is currently in development.**

## **Part 4: Standard Library Containers, Iterators and Algorithms [This content is currently in development.]**

This content is currently in development.

## **16. Standard Library**

### **Containers and Iterators [This content is currently in development.]**

**This content is currently in development.**

## **17. Standard Library**

### **Algorithms [This content is currently in development.]**

**This content is currently in development.**

## **Part 5: Other Topics [This content is currently in development.]**

**This content is currently in development.**

## **18. Intro to Custom Templates**

**[This content is currently in development.]**

**This content is currently in development.**

## **19. Stream I/O and C++20 Text Formatting: A Deeper Look**

**[This content is currently in  
development.]**

This content is currently in development.

## **20. Concurrent Programming; Intro to C++20 Coroutines [This content is currently in development.]**

This content is currently in development.

## **21. Bits, Characters, C Strings and `struct`s [This content is currently in development.]**

This content is currently in development.

## **22. Other Topics; A Look Toward C++23 and Contracts [This content is currently in development.]**

This content is currently in development.

## **Part 6: Appendices [This content is currently in development.]**

**This content is currently in development.**

## **Appendix A. Operator Precedence and Grouping [This content is currently in development.]**

**This content is currently in development.**

## **Appendix B. Character Set**

**[This content is currently in development.]**

**This content is currently in development.**

## **Appendix C. Fundamental Types [This content is currently in development.]**

**This content is currently in development.**

## **Appendix D. Number Systems**

**[This content is currently in development.]**

**This content is currently in development.**

## **Appendix E. Preprocessor**

**[This content is currently in development.]**

**This content is currently in development.**

## **Appendix F. C Legacy Code Topics [This content is currently in development.]**

**This content is currently in development.**

## **Appendix G. Using the Visual Studio Debugger [This content is currently in development.]**

**This content is currently in development.**

## **Appendix H. Using the GNU C++ Debugger [This content is currently in development.]**

**This content is currently in development.**

## **Appendix I. Using the Xcode Debugger [This content is currently in development.]**

**This content is currently in development.**

**Paul Deitel**, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses worldwide to clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

**Dr. Harvey Deitel**, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with 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 corporate, academic, government and military clients.

## The Professional Programmer's Deitel® Guide to Modern C++ Using C++20, the C++ Standard Library, Open-Source Libraries and More

The C++ programming language is popular for developing systems software, embedded systems, operating systems, real-time systems, games, communications systems and other high-performance computer applications. *C++20 for Programmers* is an introductory-through-intermediate-level, tutorial presentation of Modern C++, which consists of the four most recent C++ standards: C++11, C++14, C++17 and C++20.

Written for programmers with a background in another high-level language, *C++20 for Programmers* applies the Deitel signature **live-code approach** to teaching Modern C++ and explores the **C++20 language and libraries** in depth. The book presents concepts in fully tested programs, complete with code walkthroughs, syntax coloring, code highlighting and program outputs. It features hundreds of complete C++20 programs with thousands of lines of proven code, and hundreds of software-development tips with a special focus on performance and security, that will help you build robust applications.

Start with C++ fundamentals and the Deitels' classic treatment of object-oriented programming—classes, inheritance, polymorphism, operator overloading and exception handling. Then discover additional topics, including:

- Functional-style programming and lambdas
- Concurrency and parallelism for optimal multi-core and big data performance
- The Standard Template Library's containers, iterators and algorithms, upgraded to C++20
- Text files, CSV files, JSON serialization
- Defining custom function templates and class templates

Along the way, you'll learn compelling new **C++20** features, including **modules**, **concepts**, **ranges**, **coroutines** and Python-style **text formatting**. When you're finished, you'll have

everything you need to build industrial-strength, object-oriented C++ applications.

### Keep in Touch with the Authors

- Contact the authors at: [deitel@deitel.com](mailto:deitel@deitel.com)
- Join the Deitel social media communities:
  - LinkedIn® at <https://bit.ly/DeitelLinkedIn>
  - Facebook® at <https://facebook.com/DeitelFan>
  - Twitter® at <https://twitter.com/deitel>
  - YouTube™ at <https://youtube.com/DeitelTV>
- For source code and updates, visit: <https://deitel.com/c-plus-plus-20-for-programmers>