



DEITEL® DEVELOPER SERIES

Concepts

Standard Library

Concurrency

Coroutines

Security

Functional-Style Programming

Modules

Apple Xcode/Clang

C++ for Programmers

Ranges

STL/Parallel STL

Visualization

Lambdas

Performance

Modern C++

Open Source Libraries

Text Formatting

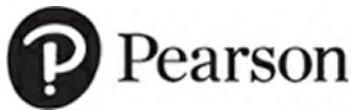
PAUL DEITEL • HARVEY DEITEL

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.

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

Part 2: Arrays, Pointers, Strings and Files

- Chapter 6. `arrays`, `vectors`, C++20 Ranges and Functional-Style Programming
- Chapter 7. (Downplaying) Pointers in Modern C++
- Chapter 8. `strings`, `string_views`, Text Files, CSV Files and Regex

Part 3: Object-Oriented Programming

- Chapter 9. Custom Classes
- Chapter 10. OOP: Inheritance and Runtime Polymorphism
- Chapter 11. Operator Overloading, Copy/Move Semantics and Smart Pointers
- Chapter 12. Exceptions and a Look Forward to Contracts

Part 4: Standard Library Containers, Iterators and Algorithms

- Chapter 13. Standard Library Containers and Iterators
- Chapter 14. Standard Library Algorithms and C++20 Ranges & Views

Part 5: Advanced Topics

- Chapter 15. Templates, C++20 Concepts and Metaprogramming
- Chapter 16. C++20 Modules: Large-Scale Development

Chapter 17. Concurrent Programming; Intro to C++20 Coroutines

Part 6: Other Topics

Chapter 18. Stream I/O; C++20 Text Formatting: A Deeper Look

Chapter 19. Other Topics; A Look Toward C++23 and Contracts

Part 7: Appendices

Appendix A: Operator Precedence and Grouping

Appendix B: Character Set

Appendix C: Fundamental Types

Appendix D: Number Systems

Appendix E: Preprocessor

Appendix F: Bits, Characters, C Strings and structs

Appendix G: C Legacy Code Topics

Appendix H: Using the Visual Studio Debugger

Appendix I: Using the GNU C++ Debugger

Appendix J: Using the Xcode Debugger

Table of Contents

Preface

Part 1: C++ Fundamentals Quickstart

Chapter 1. Introduction and Test-Driving a C++ Application

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

Chapter 2. Introduction 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

Chapter 3. Control Statements, Part 1; Intro to C++20 Text Formatting

- 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

Chapter 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

Chapter 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 Xndmwwqhlz
- 5.23 Wrap-Up

Part 2: Arrays, Pointers, Strings and Files

Chapter 6. `arrays`, `vectors`, C++20 Ranges and Functional-Style Programming

- 6.1 Introduction
- 6.2 arrays
- 6.3 Declaring arrays
- 6.4 Initializing array Elements in a Loop
- 6.5 Initializing an array with an Initializer List
- 6.6 C++11 Range-Based for and C++20 Range-Based for with Initializer
- 6.7 Setting array Elements with Calculations; Introducing `constexpr`
- 6.8 Totaling array Elements
- 6.9 Using a Primitive Bar Chart to Display array Data

- Graphically
- 6.10 Using array Elements as Counters
- 6.11 Using arrays to Summarize Survey Results
- 6.12 Sorting and Searching arrays
- 6.13 Multidimensional arrays
- 6.14 Intro to Functional-Style Programming
- 6.15 Objects Natural Case Study: C++ Standard Library Class
Template vector
- 6.16 Wrap-Up

Chapter 7. (Downplaying) Pointers in Modern C++

- 7.1 Introduction
- 7.2 Pointer Variable Declarations and Initialization
- 7.3 Pointer Operators
- 7.4 Pass-by-Reference with Pointers
- 7.5 Built-In Arrays
- 7.6 C++20: Using `to_array` to convert a Built-in Array to a
`std::array`
- 7.7 Using `const` with Pointers and the Data They Point To
- 7.8 `sizeof` Operator
- 7.9 Pointer Expressions and Pointer Arithmetic
- 7.10 Objects Natural Case Study: C++20 `spans`—Views of
Contiguous Container Elements
- 7.11 A Brief Intro to Pointer-Based Strings
- 7.12 Looking Ahead to Other Pointer Topics
- 7.13 Wrap-Up

Chapter 8. `strings`, `string_views`, Text Files, CSV Files and Regex

- 8.1 Introduction
- 8.2 `string` Assignment and Concatenation
- 8.3 Comparing `strings`
- 8.4 Substrings
- 8.5 Swapping `strings`
- 8.6 `string` Characteristics

- 8.7 Finding Substrings and Characters in a `string`
- 8.8 Replacing Characters in a `string`
- 8.9 Inserting Characters into a `string`
- 8.10 C++11 Numeric Conversions
- 8.11 C++17 `string_view`
- 8.12 Files and Streams
- 8.13 Creating a Sequential File
- 8.14 Reading Data from a Sequential File
- 8.15 C++14 Reading and Writing Quoted Text
- 8.16 Updating Sequential Files
- 8.17 String Stream Processing
- 8.18 Raw String Literals
- 8.19 Objects Natural Case Study: Reading and Analyzing a CSV File Containing Titanic Disaster Data
- 8.20 Objects Natural Case Study: Introduction to Regular Expressions
- 8.21 Wrap-Up

Part 3: Object-Oriented Programming

Chapter 9. Custom Classes

- 9.1 Introduction
- 9.2 Test-Driving an Account Object
- 9.3 Account Class with a Data Member and Set and Get Member Functions
- 9.4 Account Class: Custom Constructors
- 9.5 Software Engineering with Set and Get Member Functions
- 9.6 Account Class with a Balance
- 9.7 Time Class Case Study: Separating Interface from Implementation
- 9.8 Compilation and Linking Process
- 9.9 Class Scope and Accessing Class Members
- 9.10 Access Functions and Utility Functions
- 9.11 Time Class Case Study: Constructors with Default Arguments

- 9.12 Destructors
- 9.13 When Constructors and Destructors Are Called
- 9.14 Time Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a private Data Member
- 9.15 Default Assignment Operator
- 9.16 const Objects and const Member Functions
- 9.17 Composition: Objects as Members of Classes
- 9.18 friend Functions and friend Classes
- 9.19 The this Pointer
- 9.20 static Class Members—Classwide Data and Member Functions
- 9.21 Aggregates in C++20
- 9.22 Objects Natural Case Study: Serialization with JSON
- 9.23 Wrap-Up

Chapter 10. OOP: Inheritance and Runtime Polymorphism

- 10.1 Introduction
- 10.2 Base Classes and Derived Classes
- 10.3 Relationship between Base and Derived Classes
- 10.4 Constructors and Destructors in Derived Classes
- 10.5 Intro to Runtime Polymorphism: Polymorphic Video Game
- 10.6 Relationships Among Objects in an Inheritance Hierarchy
- 10.7 Virtual Functions and Virtual Destructors
- 10.8 Abstract Classes and Pure virtual Functions
- 10.9 Case Study: Payroll System Using Runtime Polymorphism
- 10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”
- 10.11 Non-Virtual Interface (NVI) Idiom
- 10.12 Program to an Interface, Not an Implementation
- 10.13 Runtime Polymorphism with std::variant and std::visit
- 10.14 Multiple Inheritance
- 10.15 protected Class Members

- 10.16 public, protected and private Inheritance
- 10.17 Wrap-Up

Chapter 11. Operator Overloading, Copy/Move Semantics and Smart Pointers

- 11.1 Introduction
- 11.2 Using the Overloaded Operators of Standard Library Class string
- 11.3 Operator Overloading Fundamentals
- 11.4 (Downplaying) Dynamic Memory Management with new and delete
- 11.5 Modern C++ Dynamic Memory Management—RAII and Smart Pointers
- 11.6 MyArray Case Study: Crafting a Valuable Class with Operator Overloading
- 11.7 C++20 Three-Way Comparison Operator ($\langle=\rangle$)
- 11.8 Converting Between Types
- 11.9 explicit Constructors and Conversion Operators
- 11.10 Overloading the Function Call Operator ()
- 11.11 Wrap-Up

Chapter 12. Exceptions and a Look Forward to Contracts

- 12.1 Introduction
- 12.2 Exception-Handling Flow of Control; Defining an Exception Class
- 12.3 Exception Safety Guarantees and noexcept
- 12.4 Rethrowing an Exception
- 12.5 Stack Unwinding and Uncaught Exceptions
- 12.6 When to Use Exception Handling
- 12.7 Constructors, Destructors and Exception Handling
- 12.8 Processing new Failures
- 12.9 Standard Library Exception Hierarchy
- 12.10 C++’s Alternative to the finally Block
- 12.11 Libraries Often Support Both Exceptions and Error Codes

- 12.12 Logging
- 12.13 Looking Ahead to Contracts
- 12.14 Wrap-Up

Part 4: Standard Library Containers, Iterators and Algorithms

Chapter 13. Standard Library Containers and Iterators

- 13.1 Introduction
- 13.2 Introduction to Containers
- 13.3 Working with Iterators
- 13.4 A Brief Introduction to Algorithms
- 13.5 Sequence Containers
- 13.6 vector Sequence Container
- 13.7 list Sequence Container
- 13.8 deque Sequence Container
- 13.9 Associative Containers
- 13.10 Container Adaptors
- 13.11 bitset Near Container
- 13.12 Optional: A Brief Intro to Big O
- 13.13 Optional: A Brief Intro to Hash Tables
- 13.14 Wrap-Up

Chapter 14. Standard Library Algorithms and C++20 Ranges & Views

- 14.1 Introduction
- 14.2 Algorithm Requirements: C++20 Concepts
- 14.3 Lambdas and Algorithms
- 14.4 Algorithms
- 14.5 Function Objects (Functors)
- 14.6 Projections
- 14.7 C++20 Views and Functional-Style Programming
- 14.8 Intro to Parallel Algorithms
- 14.9 Standard Library Algorithm Summary
- 14.10 A Look Ahead to C++23 Ranges
- 14.11 Wrap-Up

Part 5: Advanced Topics

Chapter 15. Templates, C++20 Concepts and Metaprogramming

- 15.1 Introduction
- 15.2 Custom Class Templates and Compile-Time Polymorphism
- 15.3 C++20 Function Template Enhancements
- 15.4 C++20 Concepts: A First Look
- 15.5 Type Traits
- 15.6 C++20 Concepts: A Deeper Look
- 15.7 Testing C++20 Concepts with `static_assert`
- 15.8 Creating a Custom Algorithm
- 15.9 Creating a Custom Container and Iterators
- 15.10 Default Arguments for Template Type Parameters
- 15.11 Variable Templates
- 15.12 Variadic Templates and Fold Expressions
- 15.13 Template Metaprogramming
- 15.14 Wrap-Up

Chapter 16. C++20 Modules: Large-Scale Development

- 16.1 Introduction
- 16.2 Compilation and Linking Prior to C++20
- 16.3 Advantages and Goals of Modules
- 16.4 Example: Transitioning to Modules—Header Units
- 16.5 Example: Creating and Using a Module
- 16.6 Global Module Fragment
- 16.7 Separating Interface from Implementation
- 16.8 Partitions
- 16.9 Additional Modules Examples
- 16.10 Modules Can Reduce Translation Unit Sizes and Compilation Times
- 16.11 Migrating Code to Modules
- 16.12 Future of Modules and Modules Tooling
- 16.13 Wrap-Up

Chapter 17. Concurrent Programming; Intro to C++20

Coroutines

Part 6: Other Topics

Chapter 18. Stream I/O; C++20 Text Formatting: A Deeper Look

Chapter 19. Other Topics; A Look Toward C++23 and Contracts

Part 7: Appendices

Appendix A: Operator Precedence and Grouping

Appendix B: Character Set

Appendix C: Fundamental Types

Appendix D: Number Systems

Appendix E: Preprocessor

Appendix F: Bits, Characters, C Strings and structs

Appendix G: C Legacy Code Topics

Appendix H: Using the Visual Studio Debugger

Appendix I: Using the GNU C++ Debugger

Appendix J: 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.

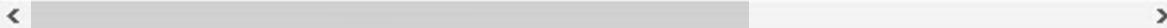
“Rough-Cut” E-Book for O’Reilly Online Learning Subscribers

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 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-livelessons>



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



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.

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

Part 2: Arrays, Pointers, Strings and Files

6. `array`, `vectors`, C++20 Ranges and Functional-Style Programming
7. (Downplaying) Pointers in Modern C++
8. `string`, Regular Expressions and Files

Part 3: Object-Oriented Programming

9. Classes
10. Inheritance and Polymorphism
11. Operator Overloading
12. Exceptions: A Deeper Look

Part 4: Standard Library Containers, Iterators and Algorithms

13. Standard Library Containers and Iterators
14. Standard Library Algorithms; Functional Programming: A Deeper Look

Part 5: Advanced Topics

15. C++20 Modules

16. Intro to Custom Templates and C++20 Concepts
17. Concurrent Programming; Intro to C++20 Coroutines

Part 6: Other Topics

18. Stream I/O; C++20 Text Formatting: A Deeper Look
19. Other Topics; A Look Toward C++23 and Contracts

Part 7: Appendices

- A. Operator Precedence and Grouping
- B. Character Set
- C. Fundamental Types
- D. Number Systems
- E. Preprocessor
- F. Bits, Characters, C Strings and structs
- G. C Legacy Code Topics
- H. Using the Visual Studio Debugger
- I. Using the GNU C++ Debugger
- J. 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

or

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

We'll respond promptly. For book updates, visit

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

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

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

This part of the quickstart guide covers the basics of C++ programming, including variables, data types, control structures, and functions.

After completing this part, you will be able to:

- Understand the basic syntax and structure of C++ code.

- Create and manipulate variables of different data types.

- Use control structures like loops and conditionals to control program flow.

- Define and use functions to reuse code and improve readability.

Estimated time: 1 hour

Prerequisites: Basic knowledge of programming concepts and experience with a text editor or IDE.

Topics covered in this part:

- Variables and Data Types

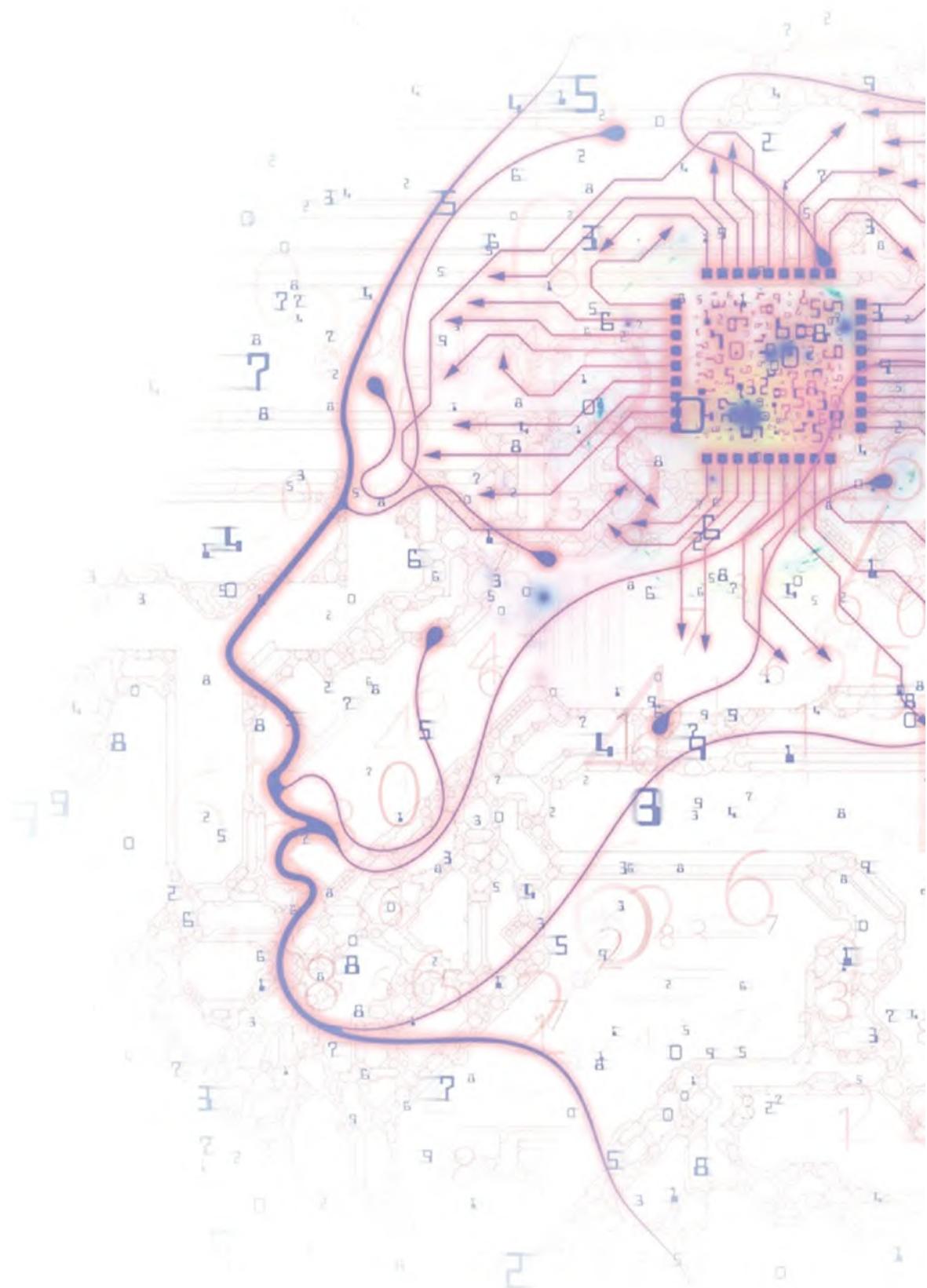
- Control Structures (Loops and Conditionals)

- Functions and Function Calls

Next steps:

Move on to Part 2: C++ Fundamentals Advanced Topics.

Chapter 1. Introduction and Test-Driving a C++ Application



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 and I'll respond promptly. Check for updates here and on the book's web page:

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

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.¹
1. 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



[“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

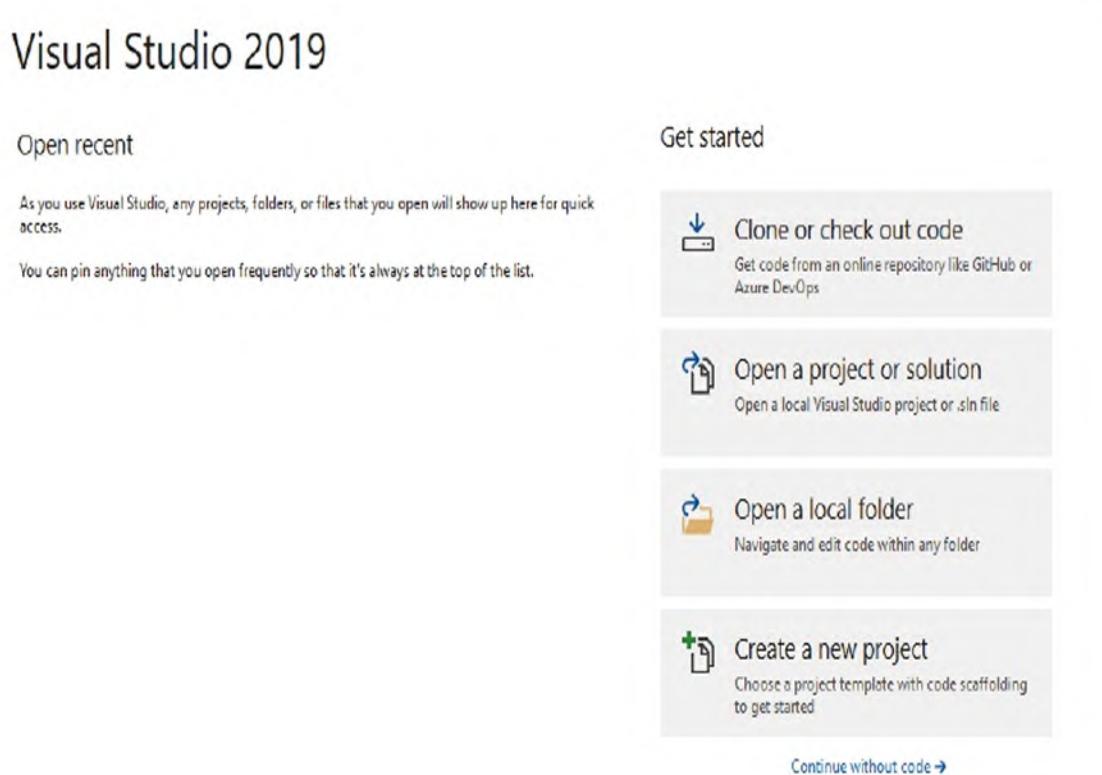
present might differ slightly. From this point forward, we'll simply say "Visual Studio" or "the IDE."

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:



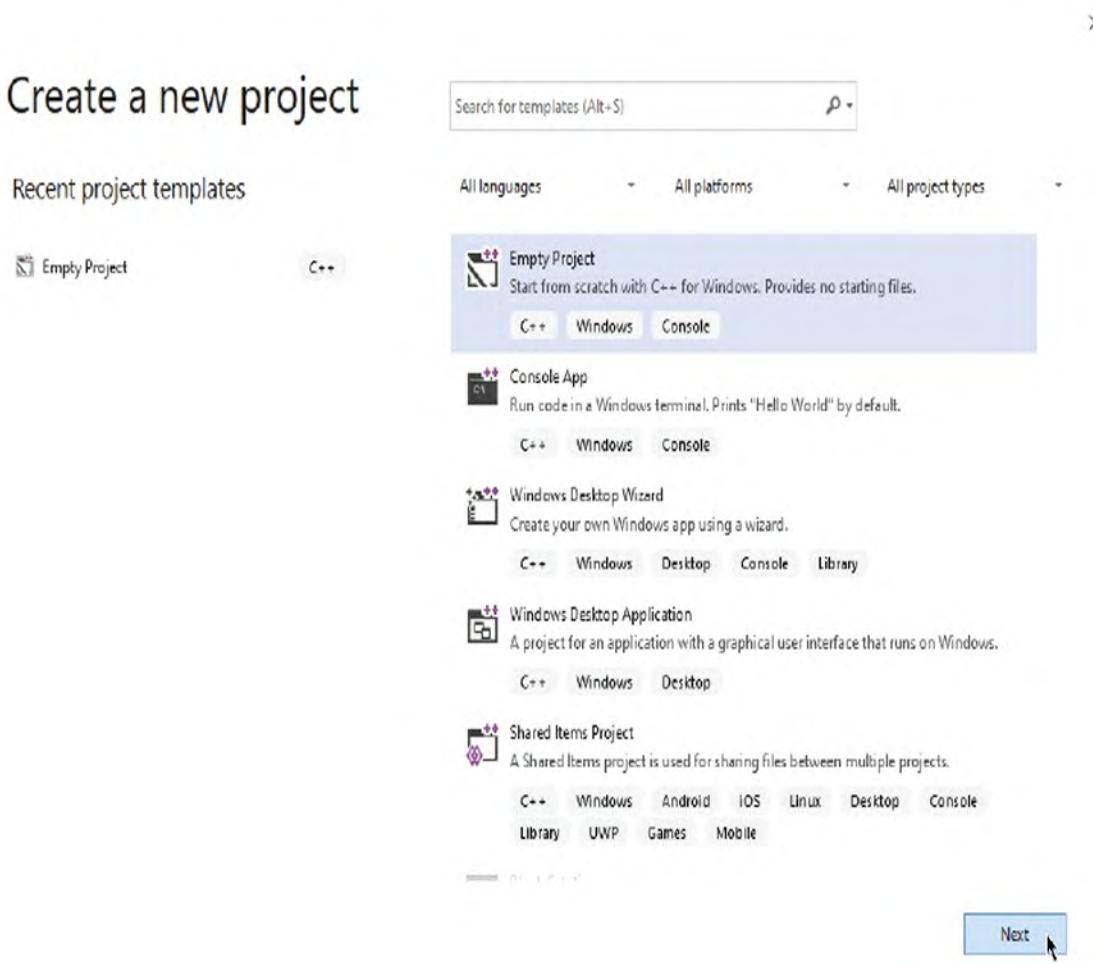
Close this window for now by clicking the **X** in its tab—you can access this window any time by selecting **File > Start Window**. We use the **>** character to indicate selecting a menu item from a menu. For example, the notation **File > Open** indicates that you should select the **Open** menu item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

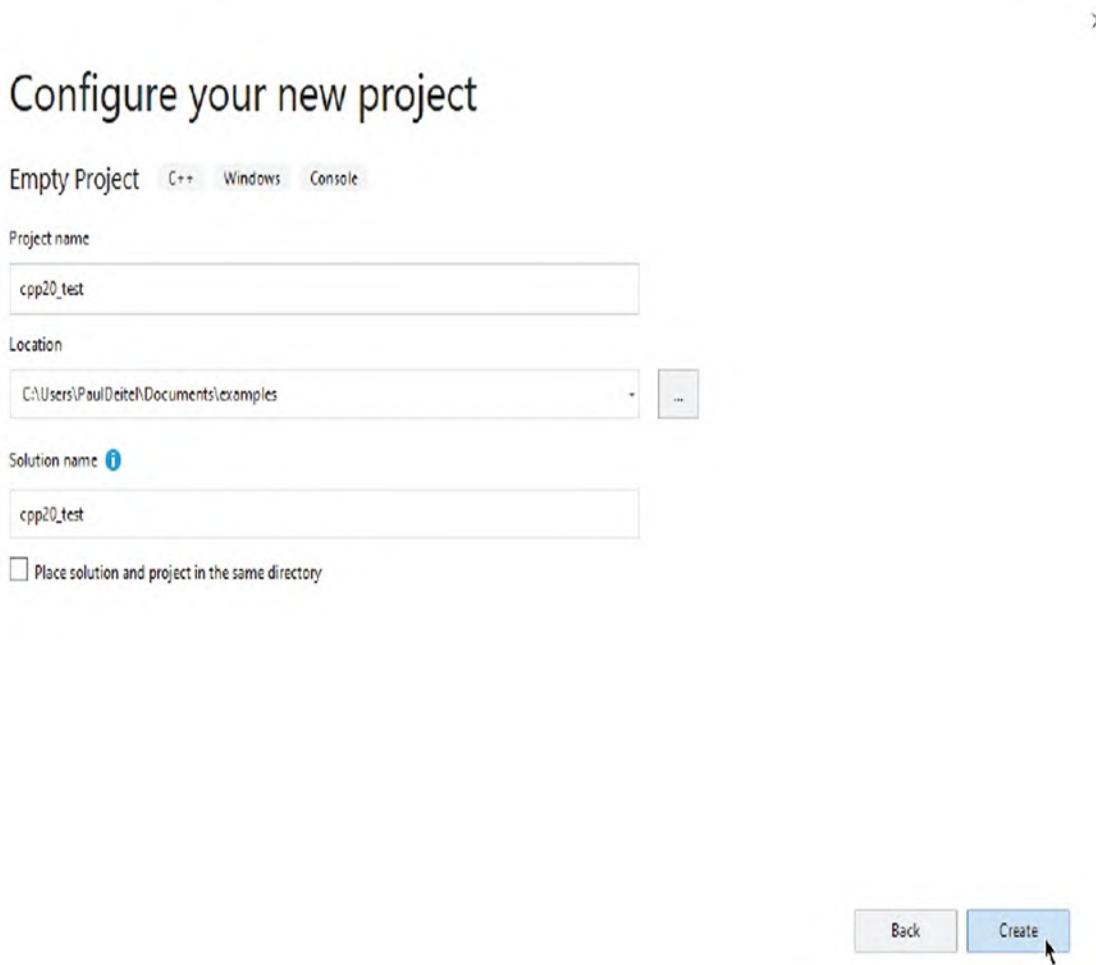
To work our code examples, you'll begin with an **Empty Project** and add files to it. To create a project:

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:

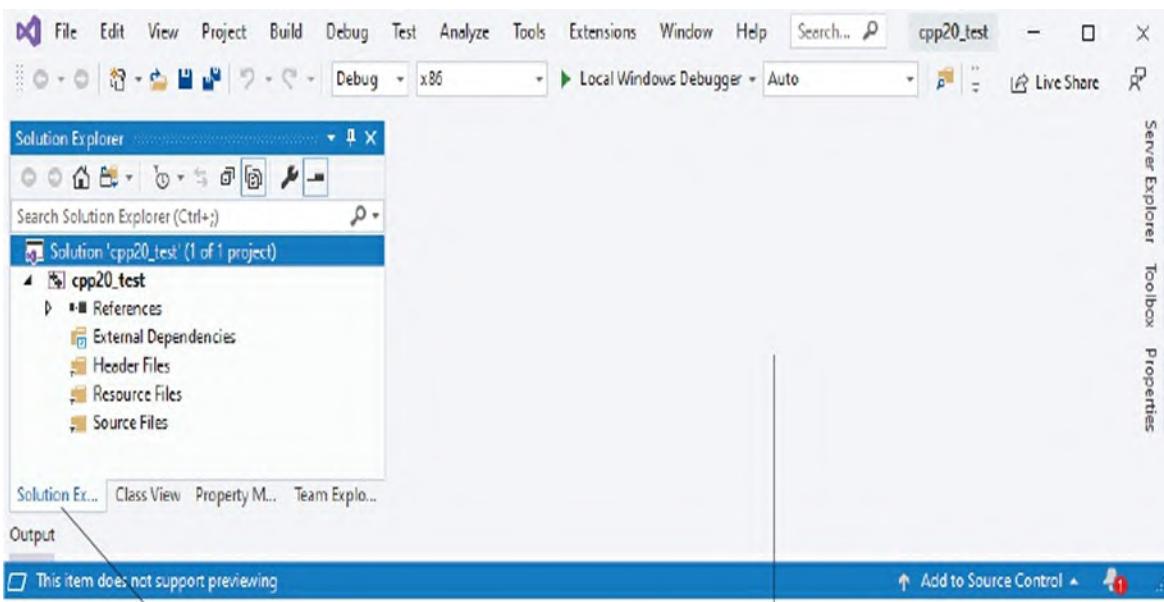


3. Provide a **Project name** and **Location** for your project. For the **Project name**, we specified `cpp20_test`. For the **Location**, we selected the `examples` folder containing this book's code examples. Click **Create** to open your new project in Visual Studio.

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:



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

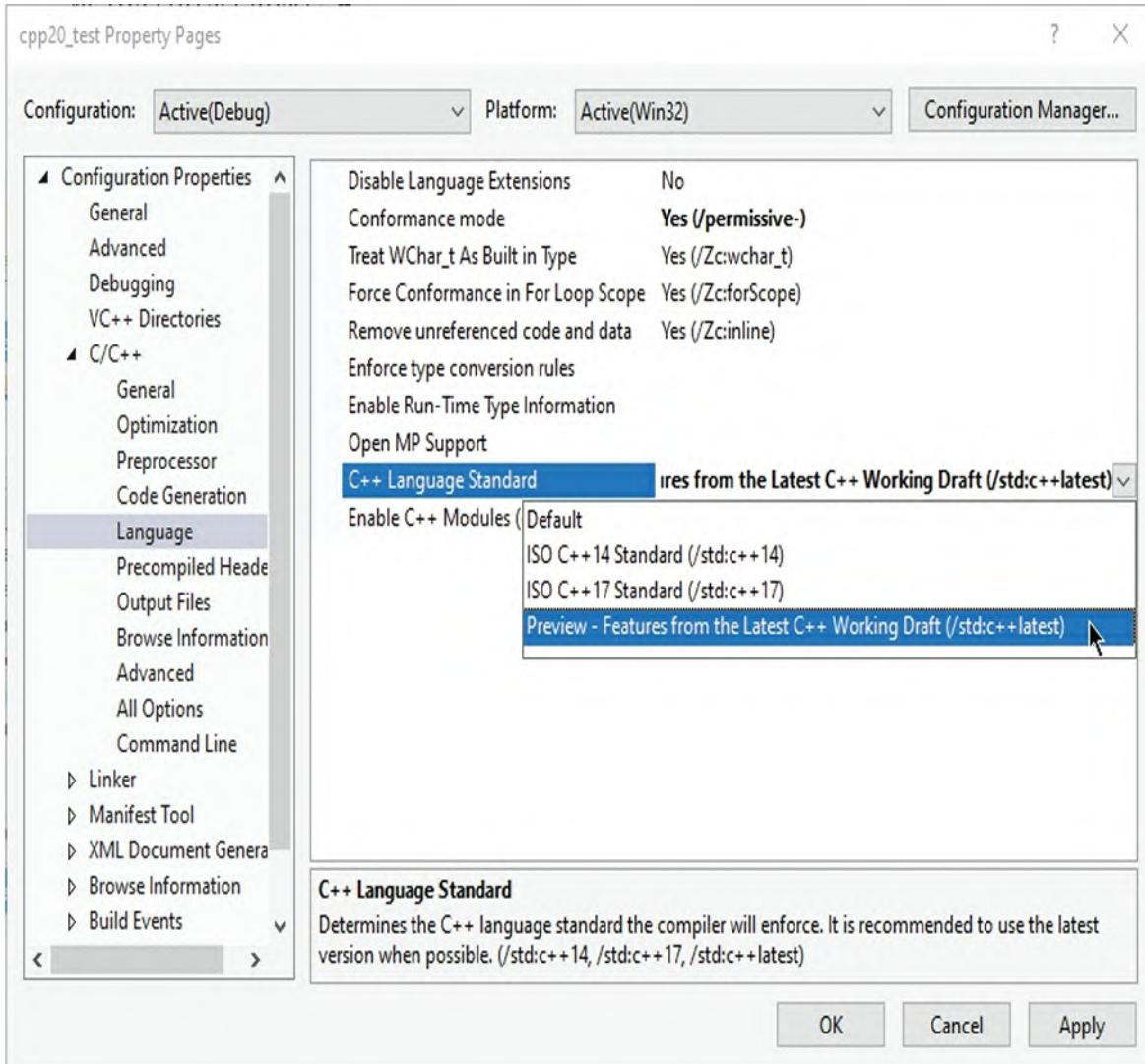
Next, you'll add `GuessNumber.cpp` to the project you created in *Step 3*. In the **Solution Explorer**:

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**.²
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:

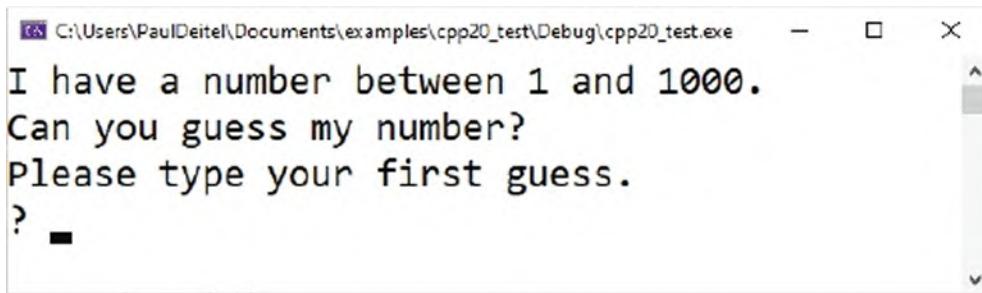


2. In the left column, expand the **C/C++** node, then select **Language**.
3. In the right column, click in the field to the right of **C++ Language Standard**, click the down arrow, then select **Preview - Features from the Latest C++ Working Draft (/std:c++latest)** and click **OK**. In a future version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

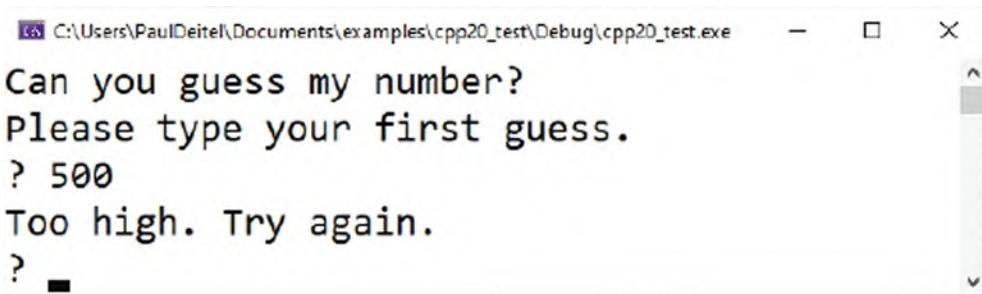


C:\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\cpp20_test.exe

```
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:

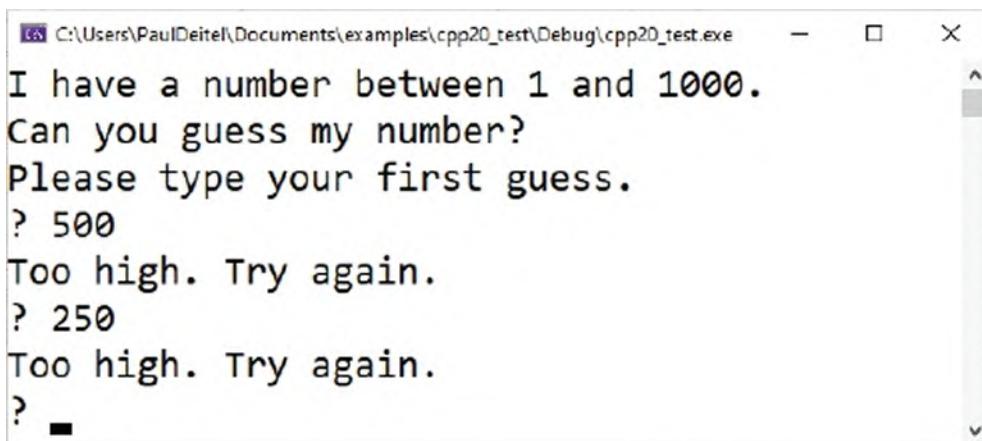


C:\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\cpp20_test.exe

```
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:

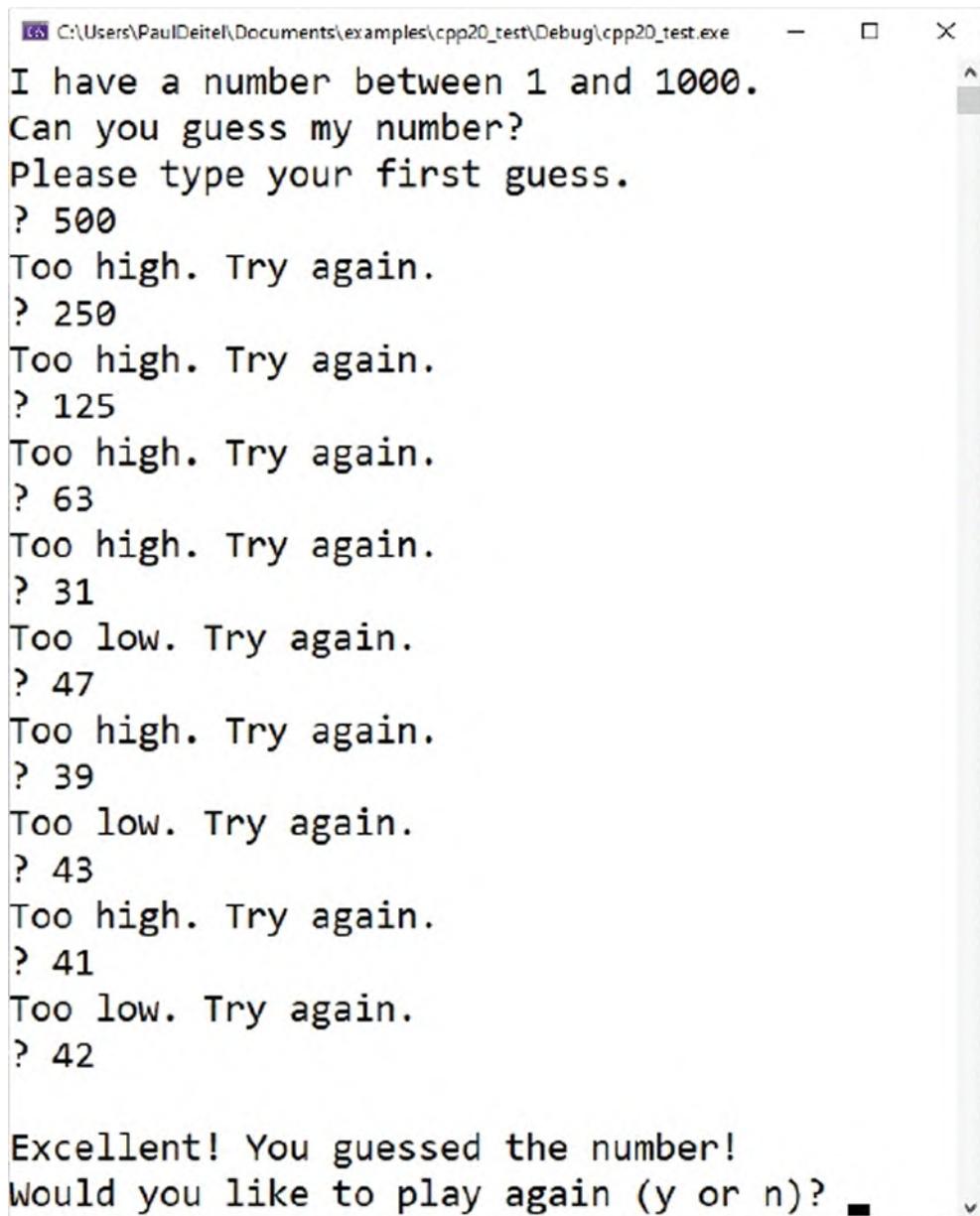


C:\Users\PaulDeitel\Documents\examples\cpp20_test\Debug\cpp20_test.exe

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
? 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 prompting the user to guess a number between 1 and 1000. The user enters '500', which is deemed 'Too high. Try again.'. Subsequent guesses of '250' and '125' are also too high. The user then guesses '63', which is 'Too high. Try again.'. This pattern repeats with guesses of '31', '47', '39', '43', '41', and '42', each being too high or too low based on the feedback provided by the application.

```
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.
? 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 (Welcome to Xcode window appears:



Welcome to Xcode

Version 11.4.1 (11E503a)

No Recent Projects



Get started with a playground
Explore new ideas quickly and easily.



Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.



Clone an existing project
Start working on something from a Git repository.

[Open another project...](#)

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

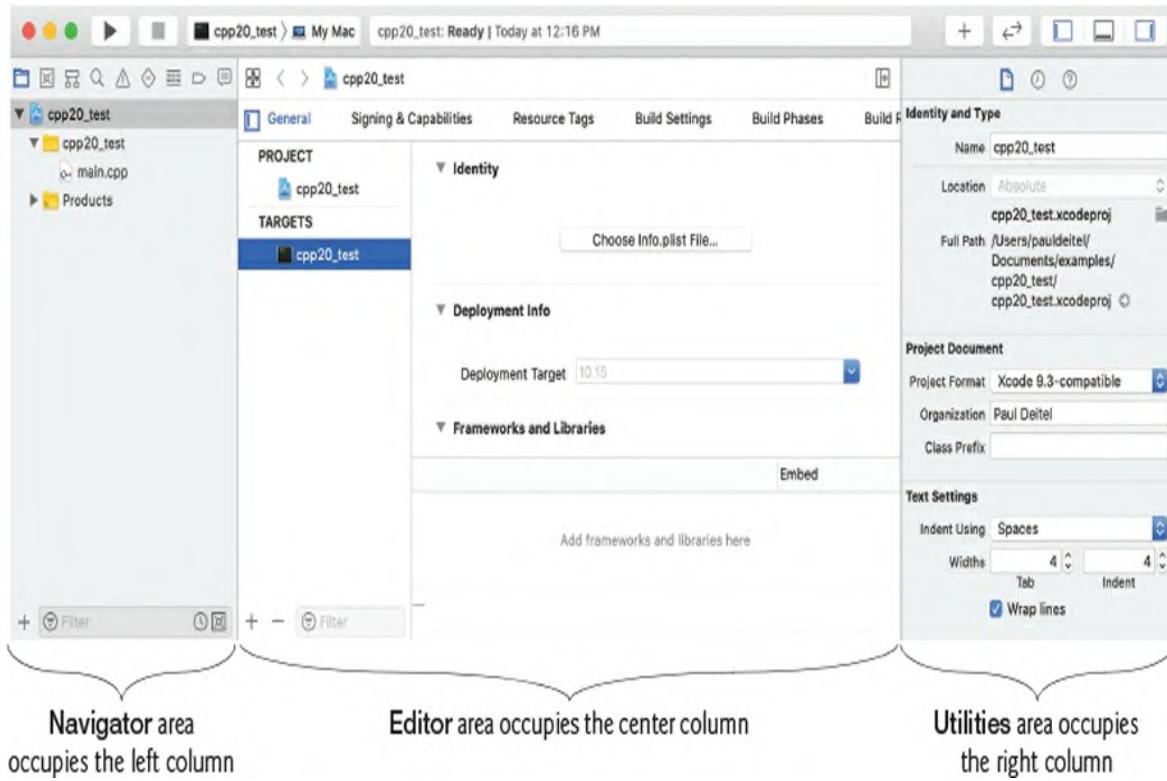
A **project** is a group of related files, such as the C++ source-code files that compose an application. The Xcode projects we created for this book’s examples are **Command Line Tool** projects that you’ll execute directly in the IDE. To create a project:

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:



Buttons for running
and stopping an app

Scheme selector
(not used in this book)

displaying the progress of tasks executing in Xcode:

cpp20_test: Ready | Today at 12:42 PM

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.

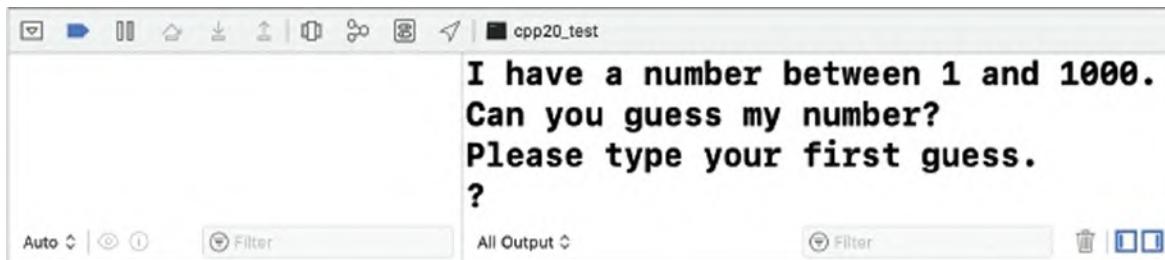
Step 6: Adding the GuessNumber.cpp File into the Project

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**.³

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:

[Click here to view code image](#)

```
~$ 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:

[Click here to view code image](#)

```
~/Documents/examples/ch01$ g++ -std=c++2a GuessNumk  
~/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:

[Click here to view code image](#)

```
~/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:

[Click here to view code image](#)

```
~/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**:

[Click here to view code image](#)

```
~/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.":

[Click here to view code image](#)

```
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

[Click here to view code image](#)

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

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

[Click here to view code image](#)

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

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. Its 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:

[Click here to view code image](#)

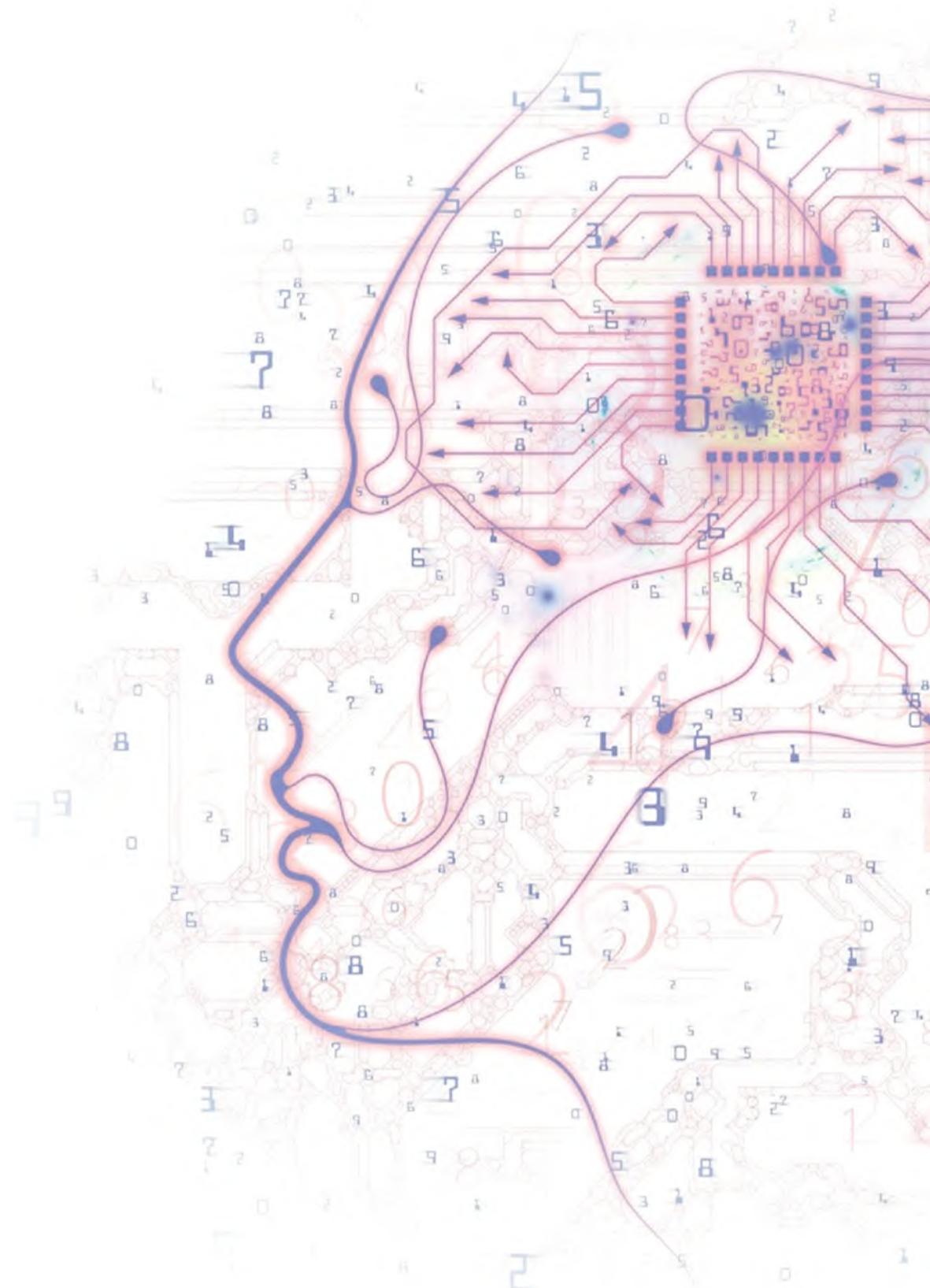
```
root@01b4d47cad6:/# cd /usr/src/ch01  
root@01b4d47cad6:/usr/src/ch01#
```

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.

Chapter 2. Introduction 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 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.

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.

[Click here to view code image](#)

```
1 // fig02_01.cpp
2 // Text-printing program.
3 #include <iostream> // enables program to output text
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 }
```

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

[Click here to view code image](#)

```
#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

[Click here to view code image](#)

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

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

The `return` Statement

Line 9

[Click here to view code image](#)

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

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.

[Click here to view code image](#)

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

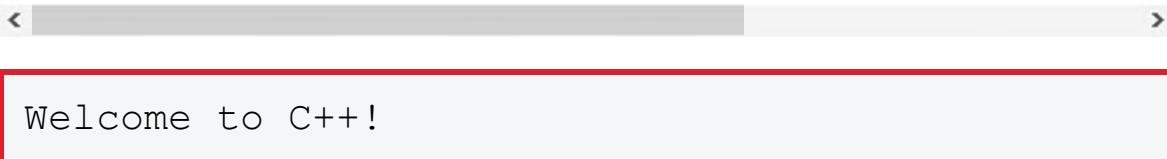


Fig. 2.2 Displaying a line of text with multiple statements.

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.

[Click here to view code image](#)

```
1 // fig02_03.cpp
2 // Displaying multiple lines of text with a single statement
3 #include <iostream> // enables program to output text
4
```

```
5 // function main begins program execution
6 int main() {
7     std::cout << "Welcome\n\tto\n\tC++!\n";
8 } // end function main
```



```
Welcome
to

C++!
```

Fig. 2.3 Displaying multiple lines of text with a single statement.

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**.

[Click here to view code image](#)

```
1 // fig02_04.cpp
2 // Addition program that displays the sum of
3 #include <iostream> // enables program to pe
4
5 // function main begins program execution
6 int main() {
7     // declaring and initializing variables
8     int number1{0}; // first integer to add (
9     int number2{0}; // second integer to add
10    int sum{0}; // sum of number1 and number2
11
12    std::cout << "Enter first integer: "; // :
13    std::cin >> number1; // read first intege
14
```

```
15     std::cout << "Enter second integer: "; //  
16     std::cin >> number2; // read second integer  
17  
18     sum = number1 + number2; // add the numbers  
19  
20     std::cout << "Sum is " << sum << std::endl;  
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

[Click here to view code image](#)

```
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)
```

< >

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:

[Click here to view code image](#)

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



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

[Click here to view code image](#)

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



could have been placed immediately before line 13

[Click here to view code image](#)

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



the declaration in line 9

[Click here to view code image](#)

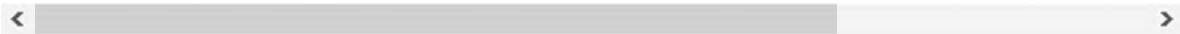
```
int number2{0}; // second integer to add (initiali
```



could have been placed immediately before line 16

[Click here to view code image](#)

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



and the declaration in line 10

[Click here to view code image](#)

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



could have been placed immediately before line 18

[Click here to view code image](#)

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



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

[Click here to view code image](#)

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



Obtaining the First Value from the User

Line 12

[Click here to view code image](#)

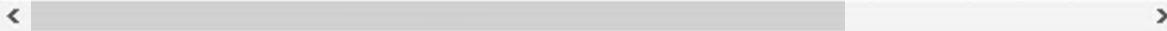
```
std::cout << "Enter first integer: "; // prompt user
```



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

[Click here to view code image](#)

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



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

[Click here to view code image](#)

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



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

[Click here to view code image](#)

```
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

[Click here to view code image](#)

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



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

[Click here to view code image](#)

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



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

[Click here to view code image](#)

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

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	*	bm 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.

[Click here to view code image](#)

```
1 // fig02_05.cpp
2 // Comparing integers using if statements, r
3 // and equality operators.
4 #include <iostream> // enables program to pe
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 comp
13     int number2{0}; // second integer to comp
14
15     cout << "Enter two integers to compare:
16     cin >> number1 >> number2; // read two i
17
18     if (number1 == number2) {
19         cout << number1 << " == " << number2
20     }
21
22     if (number1 != number2) {
23         cout << number1 << " != " << number2
24     }
25
26     if (number1 < number2) {
27         cout << number1 << " < " << number2 <
28     }
29
30     if (number1 > number2) {
31         cout << number1 << " > " << number2 <
32     }
33 }
```

```
34     if (number1 <= number2) {  
35         cout << number1 << " <= " << number2  
36     }  
37  
38     if (number1 >= number2) {  
39         cout << number1 << " >= " << number2  
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.

using Declarations

Lines 6–8

[Click here to view code image](#)

```
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.¹

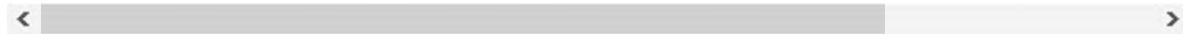
1. In Chapter 23, Other Topics, we'll discuss some issues with using directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User

Lines 12–13

[Click here to view code image](#)

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



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

Line 16

[Click here to view code image](#)

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



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

[Click here to view code image](#)

```
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 = 0` 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`² 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.

[Click here to view code image](#)

```
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 length
13    cout << "s1: \" " << s1 << "\"; length: "
14        << "\ns2: \" " << s2 << "\"; length: "
15        << "\ns3: \" " << s3 << "\"; length: "
16
17    // compare strings with == and !=
18    cout << "\n\nThe results of comparing s2 and s3 are:
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";
27        s3 = s1 + s2; // assign s3 the result of concatenation
28        cout << "s3: \" " << s3 << "\"";
29    }
```

```
30
31     // testing new C++20 string member functions
32     cout << "\n\ns1 starts with \"ha\": " << s1.substr(0, 2)
33     cout << "s2 starts with \"ha\": " << s2.substr(0, 2)
34     cout << "s1 ends with \"ay\": " << s1.substr(s1.length - 2)
35     cout << "s2 ends with \"ay\": " << s2.substr(s2.length - 2)
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.

Instantiating Objects

Typically, you cannot call a member function of a class until you create an object of that class³—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.⁴

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`

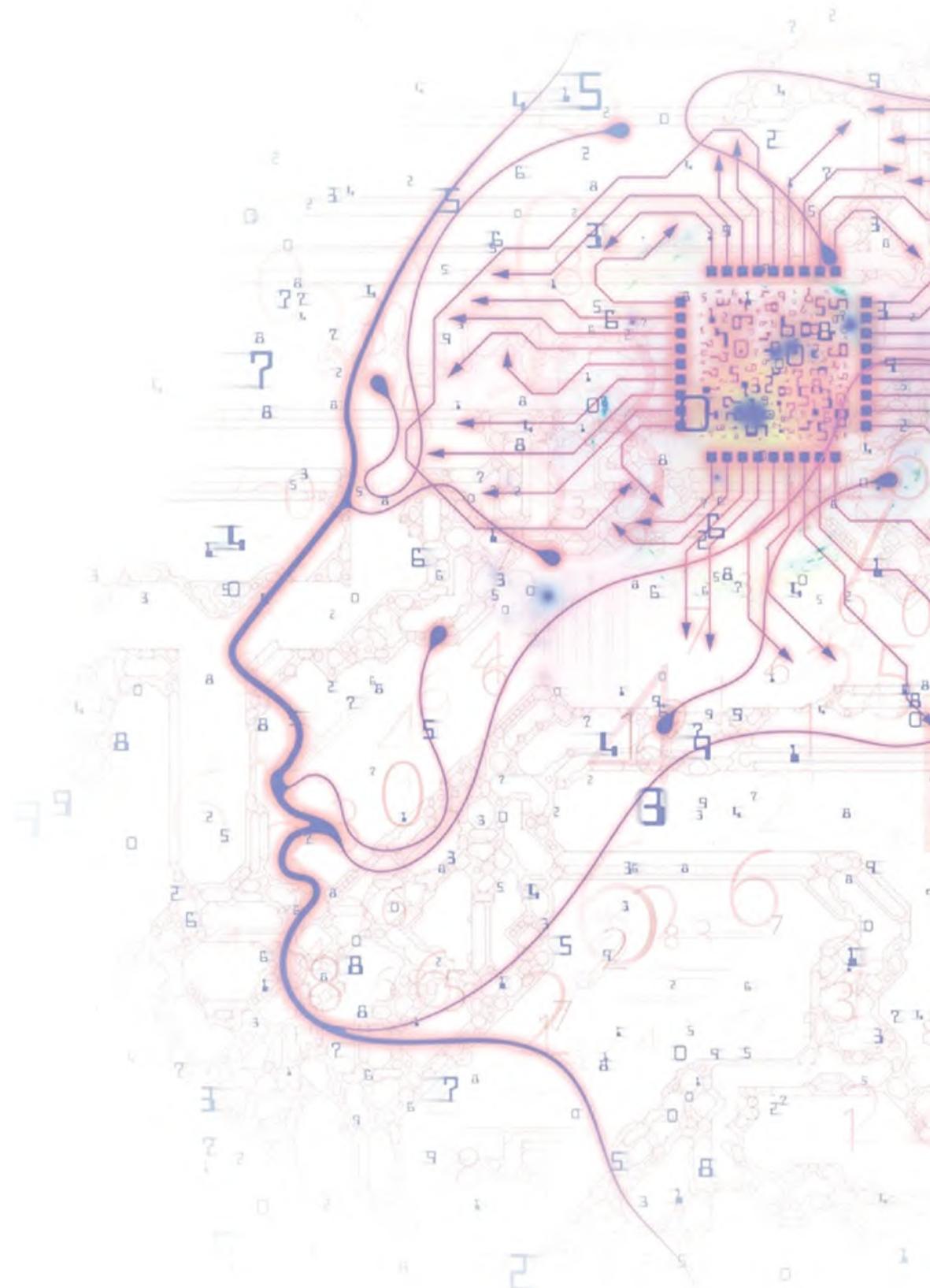
20 Lines 32–35 demonstrate new `string` member functions `starts_with` and `ends_with`, which return `true` if the string starts with or ends with a specified substring, respectively; otherwise, they return `false`. Lines 32 and 33 show that `s1` starts with "ha", but `s2` does not. Lines 34 and 35 show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

Chapter 3. Control Statements, Part 1; Intro to C++20 Text Formatting



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.¹ 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.html>.

“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 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-fundamen...>



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 Jacopini² 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`

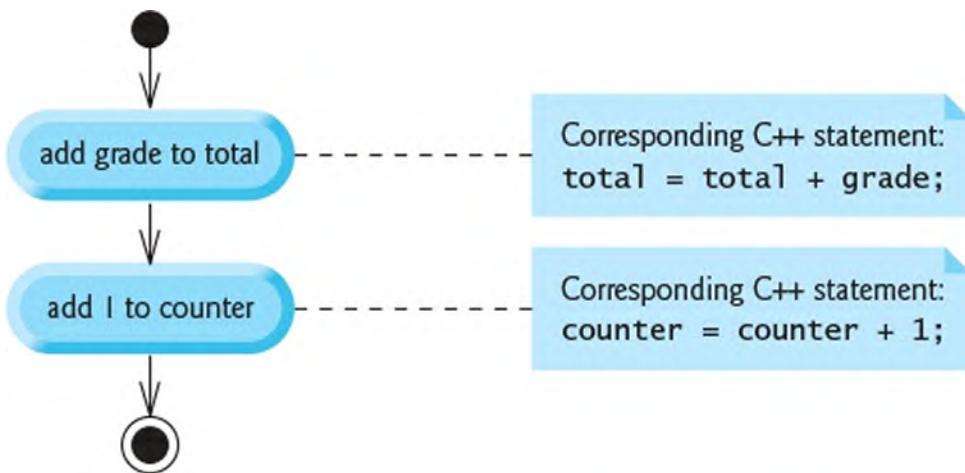
elimination.” The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and 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 UML³ **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

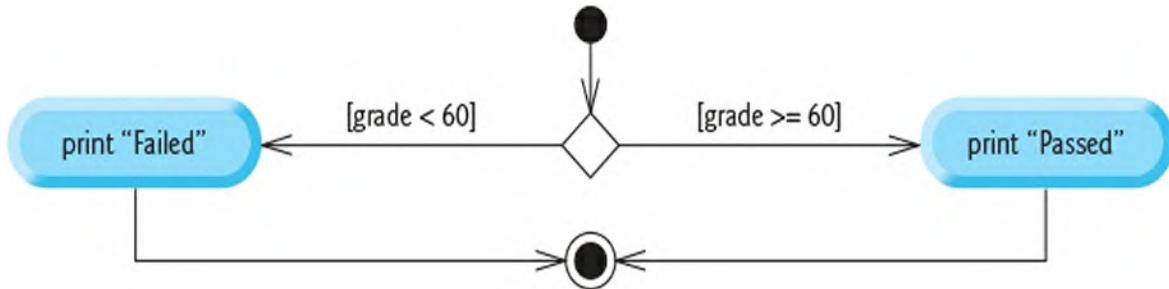
```
if (grade >= 60) {  
    cout << "Passed";  
}  
else {  
    cout << "Failed";  
}
```

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:

[Click here to view code image](#)

```

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

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
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":

[Click here to view code image](#)

```
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};

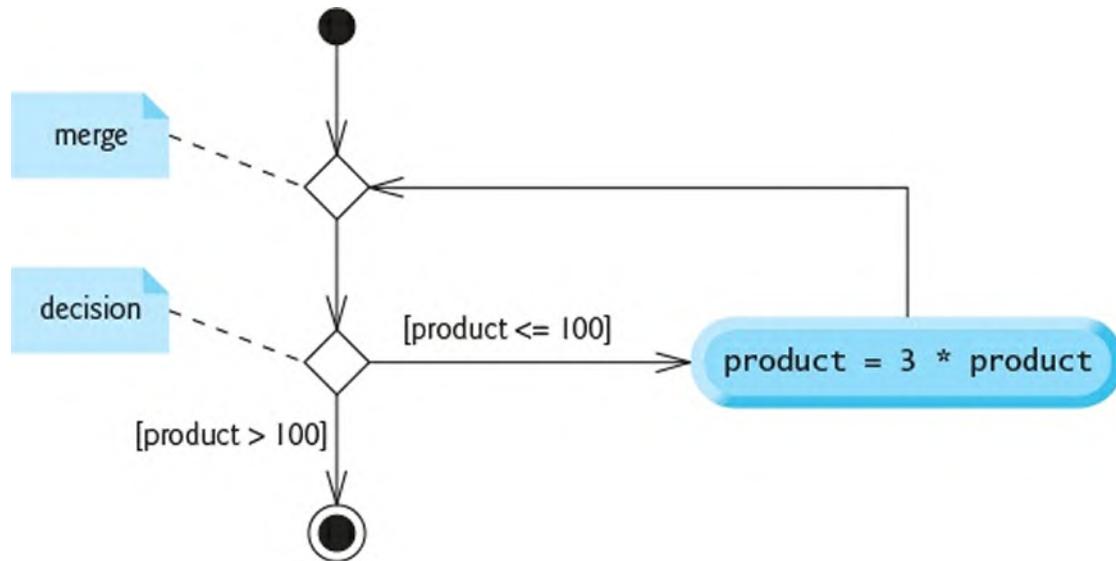
while (product <= 100) {
    product = 3 * product;
}
```

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.

[Click here to view code image](#)

```
1  fig03_01.cpp
2  // Solving the class-average problem using c
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7      // initialization phase
8      int total{0}; // initialize sum of grades
9      int gradeCounter{1}; // initialize grade
10
11     // processing phase uses counter-controll
12     while (gradeCounter <= 10) { // loop 10 t
13         cout << "Enter grade: "; // prompt
14         int grade;
15         cin >> grade; // input next grade
16         total = total + grade; // add grade to
17         gradeCounter = gradeCounter + 1; // in
18     }
19
20     // termination phase
21     int average{total / 10}; // int division
22 }
```

```
23     // display total and average of grades
24     cout << "\nTotal of all 10 grades is " <<
25     cout << "\nClass average is " << average
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.

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.

[Click here to view code image](#)

```
1 // fig03_02.cpp
2 // Solving the class-average problem using sentinel control
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
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; // increase counter
22
23        // prompt for input and read next grade
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 ca
32         double average{static_cast<double>(tot.
33
34         // display total and average (with two
35         cout << "\nTotal of the " << gradeCoun
36             << " grades entered is " << total;
37             cout << setprecision(2) << fixed;
38             cout << "\nClass average is " << avera
39     }
40     else { // no grades were entered, so outp
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.

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

[Click here to view code image](#)

```
double average{static_cast<double>(total) / gradeCounter};
```

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**⁴, 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 `3.1e+03` (that is, 3.1×10^3). 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.⁵

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.

[Click here to view code image](#)

```
1 // fig03_03.cpp
2 // Analysis of examination results using nested loops
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
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    }
24
25    // output the results
26    cout << "Number of passes: " << passes << endl;
27    cout << "Number of failures: " << failures << endl;
28
29    // determine if there is a bonus
30    if (passes > 8) {
31        cout << "Bonus to instructor!" << endl;
32    }
33}
```

```
23     else {
24         failures = failures + 1;
25     }
26
27     // increment studentCounter so loop even
28     studentCounter = studentCounter + 1;
29 }
30
31 // termination phase; prepare and display
32 cout << "Passed: " << passes << "\nFailed
33
34 // determine whether more than 8 students
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.

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 2⁶ 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!"

6. 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

[Click here to view code image](#)

```
Type 'double' cannot be narrowed to 'int' in init:
```

```
< >
```

Visual Studio gives the error

[Click here to view code image](#)

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

```
< >
```

and GNU C++ gives the error

[Click here to view code image](#)

```
type 'double' cannot be narrowed to 'int' in init:  
[-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

[Click here to view code image](#)

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



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:</i> <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>			
<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 <code>number</code> by 1, then use the new value of <code>number</code> in the expression in which <code>number</code> resides.
<code>++</code>	postfix increment	<code>number++</code>	Use the current value of <code>number</code> in the expression in which <code>number</code> resides, then increment <code>number</code> by 1.
<code>--</code>	prefix decrement	<code>--number</code>	Decrement <code>number</code> by 1, then use the new value of <code>number</code> in the expression in which <code>number</code> resides.
<code>--</code>	postfix decrement	<code>number--</code>	Use the current value of <code>number</code> in the expression in which <code>number</code> resides, then decrement <code>number</code> 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.

[Click here to view code image](#)

```
1 // fig03_04.cpp
2 // Prefix increment and postfix increment op-
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 <
10    cout << " postincrementing c: " << c++ <<
11    cout << " c after postincrement: " << c <
12
13    cout << endl; // skip a line
14
15    // demonstrate prefix increment operator
16    c = 5;
17    cout << " c before preincrement: " << c <
18    cout << " preincrementing c: " << ++c <<
19    cout << " c after preincrement: " << c <
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)

[Click here to view code image](#)

```
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
:: ()	left to right <i>[See Chapter 2's caution regarding grouping parentheses.]</i>	primary
++ -- static_cast<type>()	left to right	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
?:	right to left	conditional
= += -= *= /= %=	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.⁷ In the next section, we'll show one way to achieve portability.

7. 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.⁸ You simply include its header file (bignumber.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.

8. 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.

[Click here to view code image](#)

```
1 // fig03_05.cpp
2 // Integer ranges and arbitrary precision in
3 #include <iostream>
4 #include "bignum.h"
5 using namespace std;
6
7 int main() {
8     // use the maximum long long fundamental
9     long long value1{9'223'372'036'854'775'80
10    cout << "long long value1: " << value1
11    << "\nvalue1 - 1 = " << value1 - 1 // !
```

```

12      << "\nvalue1 + 1 = " << value1 + 1; // 
13
14 // use an arbitrary precision integer
15 BigNumber value2{value1};
16 cout << "\n\nBigNumber value2: " << value2;
17     << "\nvalue2 - 1 = " << value2 - 1 // 
18     << "\nvalue2 + 1 = " << value2 + 1; // 
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 exceed
28     cout << "\nvalue3 to the power " << co
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 }

```

long long value1: 9223372036854775807

Fig. 3.5 Integer ranges and arbitrary precision integers.

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.

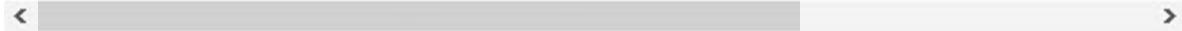
What Happens When You Exceed the Maximum long long Integer Value?

Line 9 initializes the variable `value1` with the maximum long long value on our system:⁹

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.

[Click here to view code image](#)

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



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

[Click here to view code image](#)

```
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\I`
- 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:

[Click here to view code image](#)

```
g++ -std=c++2a -I BigNumber/src fig03_05.cpp \
BigNumber/src/bignumber.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 `bignumber.h` and `bignumber.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 $\mathfrak{H} + R$ 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.¹⁰ 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.html>.

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.^{11,12} So, we'll use this library until the C++20 compilers implement text formatting.¹³ 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.html>, 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.

[Click here to view code image](#)

```
1 // fig03_06.cpp
2 // C++20 string formatting.
3 #include <iostream>
4 #include "fmt/format.h" // C++20: This will :
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);
13 }
```



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.

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 f:
```



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:

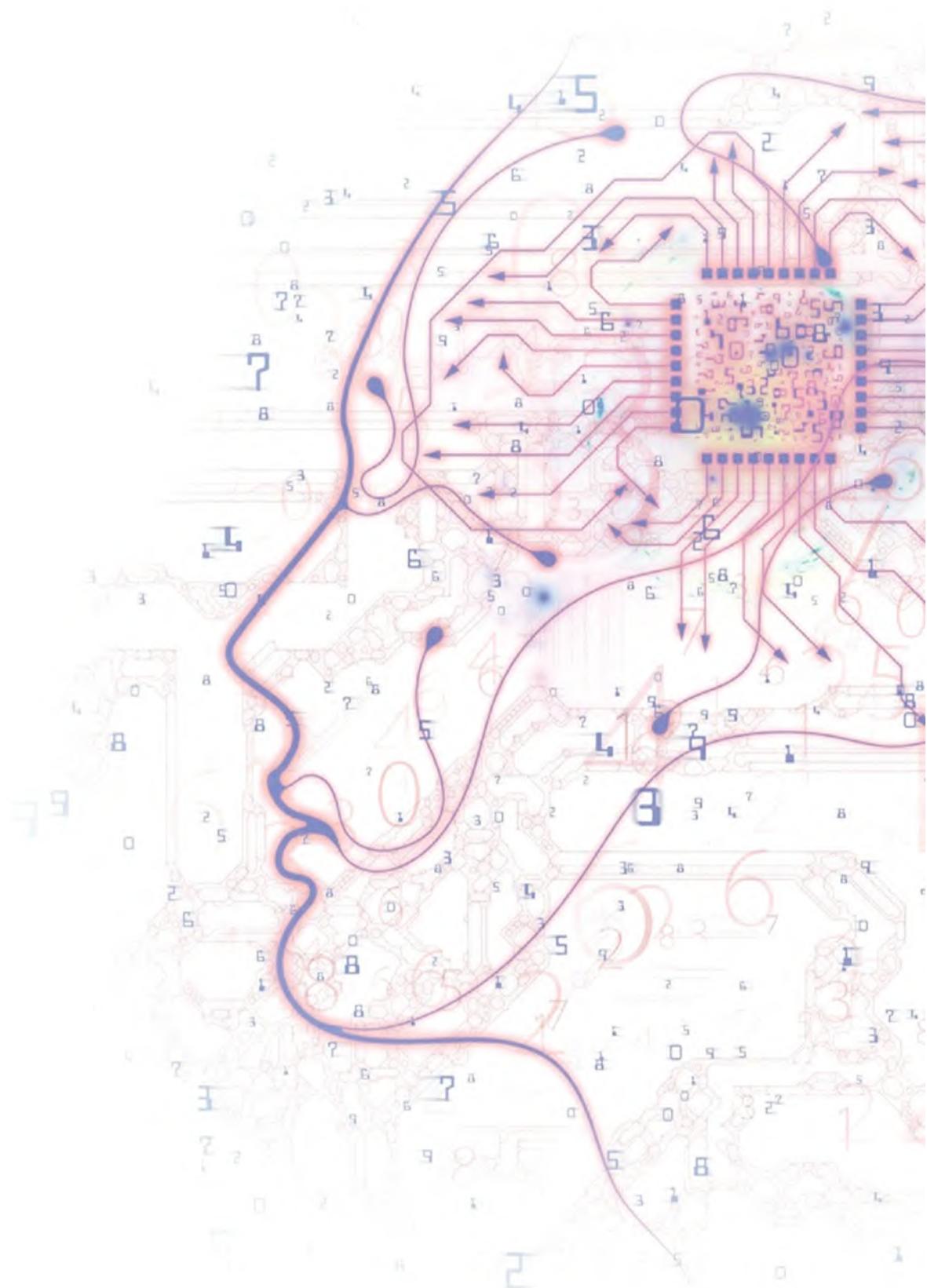
- In Step 2, drag the files `fig03_06.cpp`, `format.cc` and the folder

`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.

Chapter 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 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.

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.

[Click here to view code image](#)

```
1 // fig04_01.cpp
```

```
2 // Counter-controlled iteration with the while loop
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int counter{1}; // declare and initialize
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.

[Click here to view code image](#)

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



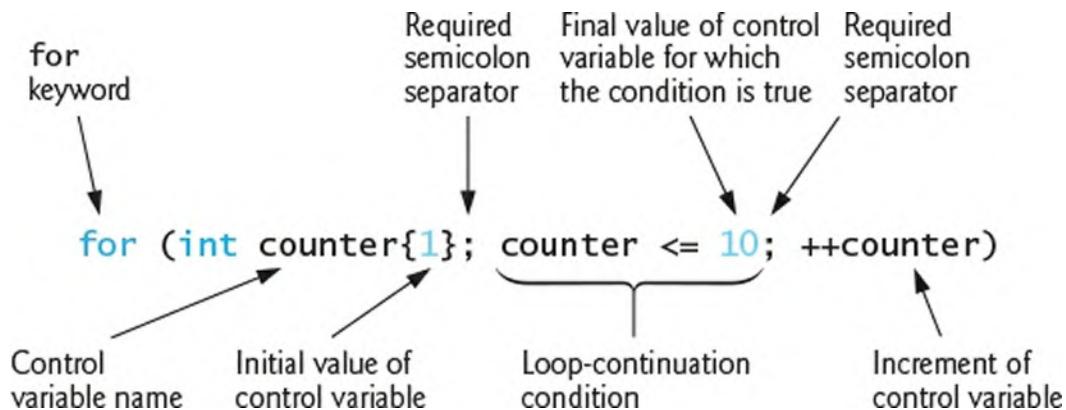
Fig. 4.2 Counter-controlled iteration with the **for** iteration statement.

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.

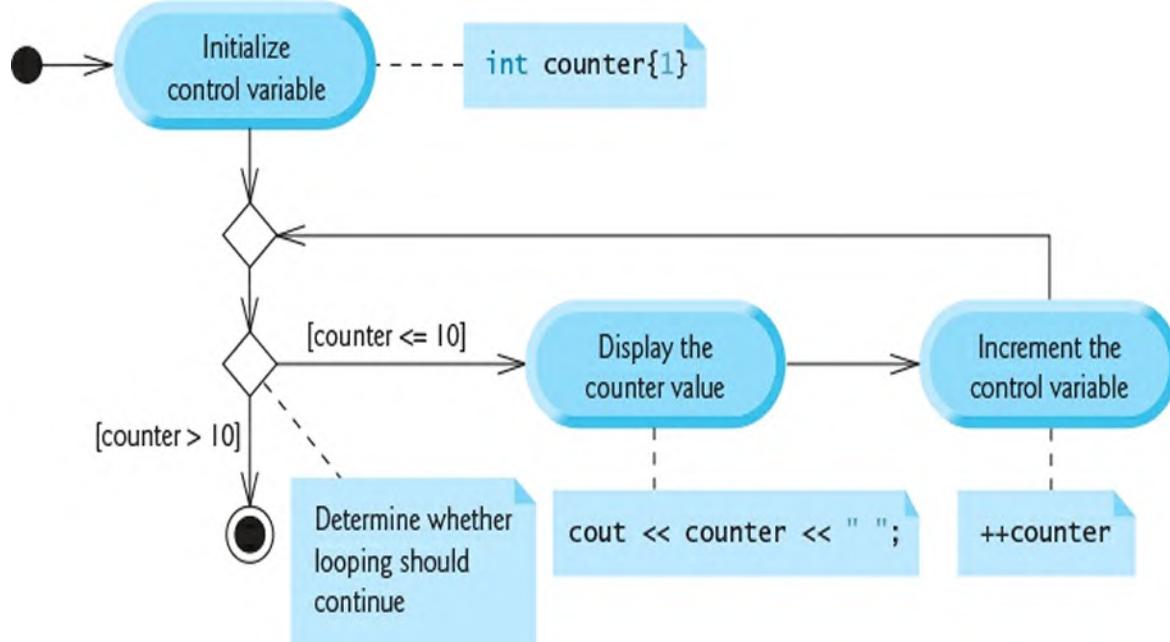
Using a `for` Statement's Control Variable in the Statement's Body

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

[Click here to view code image](#)

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

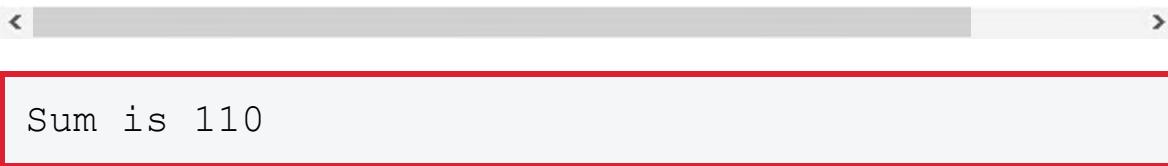
`counter != 10` never becomes false (resulting in an infinite loop) because `counter` increments by 2 after each iteration, producing only the odd values (3, 5, 7, 9, 11, ...).

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`.

[Click here to view code image](#)

```
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
11        total += number;
12    }
13
14    cout << "Sum is " << total << endl;
15 }
```



The screenshot shows a terminal window with a light gray background. At the top, there is a horizontal scrollbar with arrows on both ends. Below the scrollbar, the text "Sum is 110" is displayed in a monospaced font. The entire terminal window is enclosed in a red rectangular border.

Fig. 4.3 Summing integers with the `for` statement.

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.

w (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.

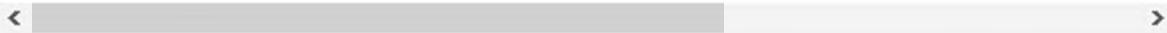
[Click here to view code image](#)

```
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 amo
13    double rate{0.05}; // interest rate
14
15    cout << "Initial principal: " << principa
16    cout << "      Interest rate: " << rate << '\n';
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount o
20
21    // calculate amount on deposit for each o
22    for (int year{1}; year <= 10; ++year) {
23        // calculate amount on deposit at the e
24        double amount = principal * pow(1.0 + rate, year);
25
26        // display the year and the amount
27        cout << setw(4) << year << setw(20) << amount << '\n';
28    }
29}

```



Initial principal: 1000.00
 Interest rate: 0.05

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51

5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

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

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`.¹ 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.

 **PERF** The body of the for statement contains the calculation $1.0 + \text{rate}$ 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;
```

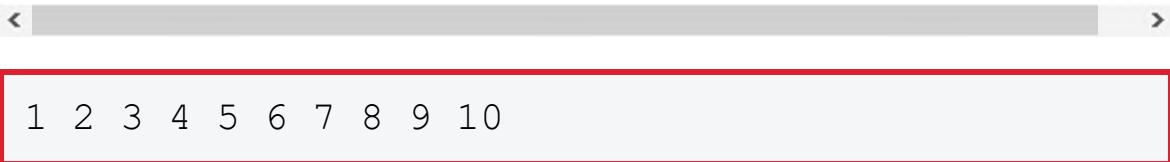
then displayed variable d's value with many digits of precision to the right of the decimal point. The resulting output showed 123.02 as 123.0199999..., which is another example of a representational error. Though some dollar amounts can be represented precisely as double, many cannot. This is a common 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.

[Click here to view code image](#)

```
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 }
```



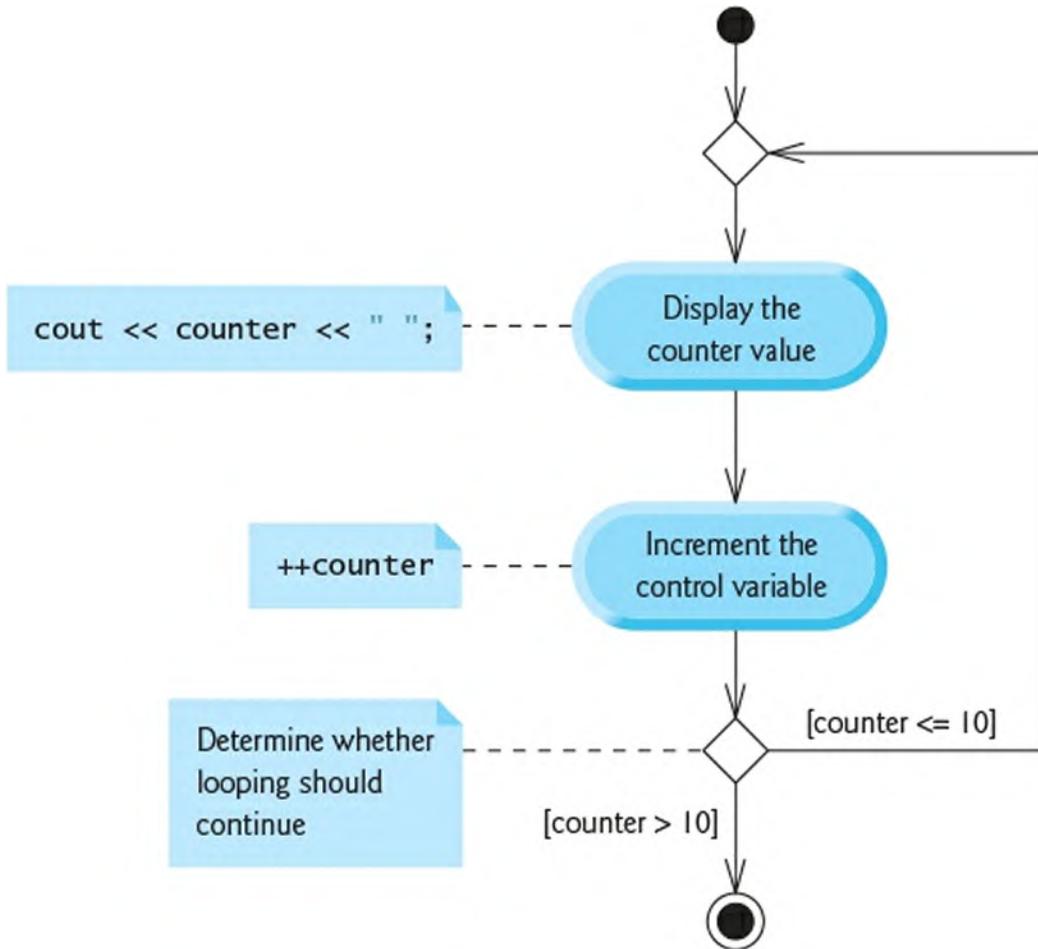
```
1 2 3 4 5 6 7 8 9 10
```

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.

[Click here to view code image](#)

```
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
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 following format:
17        << "Type the end-of-file indicator to indicate you are finished.
18        << "On UNIX/Linux/macOS type <Ctrl>D
19        << "On Windows type <Ctrl> z then press enter
20
21    int grade;
22
23    // loop until user enters the end-of-file
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 and 99, inclusive
31            case 10: // and 100, inclusive
32                ++aCount;
33                break; // exits switch
34
35            case 8: // grade was between 80 and 89, inclusive
36                ++bCount;
37                break; // exits switch
38
39        }
40    }
41
42    // output results
43    cout << "The total grade is " << total << endl;
44    cout << "The number of grades entered is " << gradeCounter << endl;
45    cout << "The number of A grades is " << aCount << endl;
46    cout << "The number of B grades is " << bCount << endl;
47}
```

```
38
39         case 7: // grade was between 70 and
40             ++cCount;
41             break; // exits switch
42
43         case 6: // grade was between 60 and
44             ++dCount;
45             break; // exits switch
46
47     default: // grade was less than 60
48         ++fCount;
49         break; // optional; exits switch
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 is "
66     << total << "\nClass average is " << average;
67     << "\nNumber of students who received an A: " << aCount << "\nB: " << bCount
68     << "\nC: " << cCount << "\nD: " << dCount << "\nF: " << fCount;
69
70 }
71 else { // no grades were entered, so output a 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.

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`.²

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

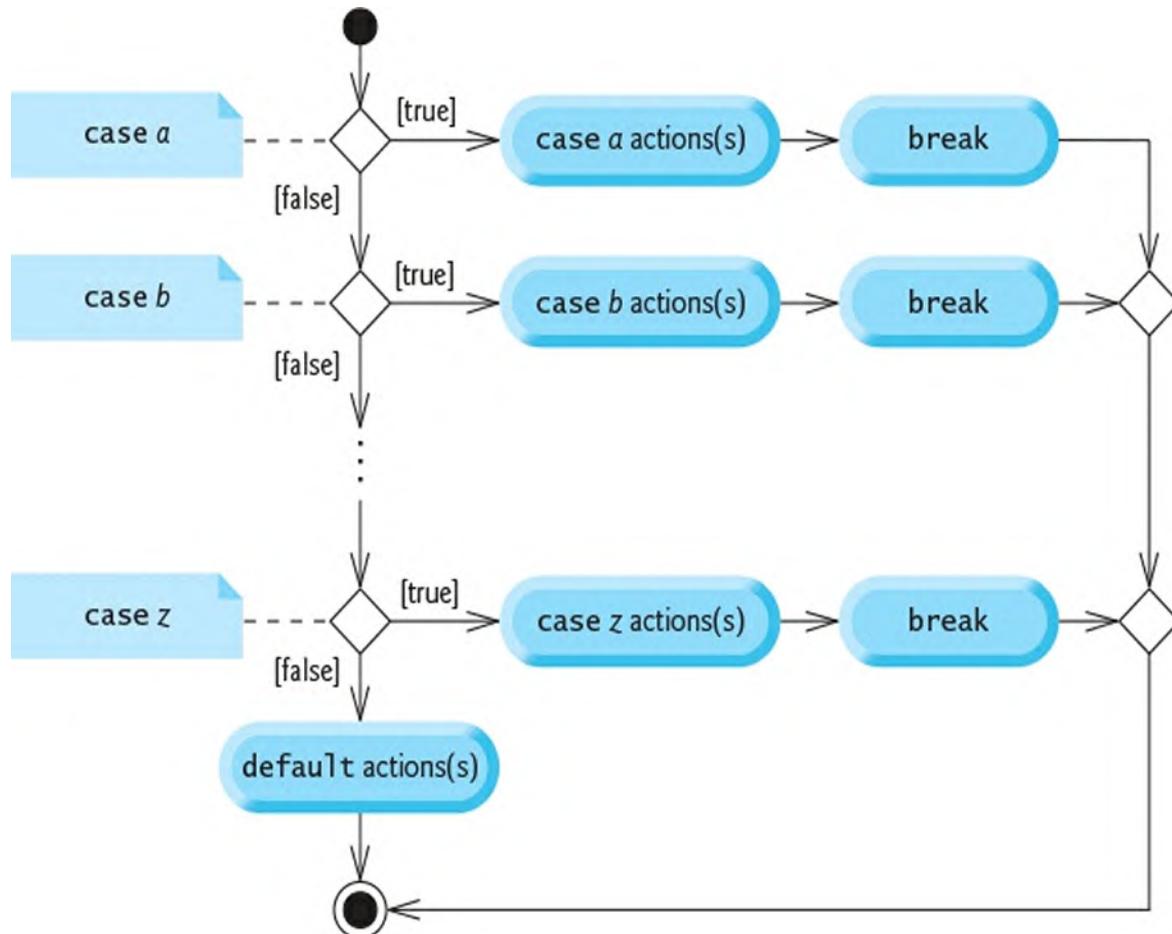
If no match occurs between the controlling expression’s value and any of the `case` labels, the `default` case (lines 47–49) executes. We use the `default` case in this example to process all controlling-expression values that are less than 6—that is, all failing grades. If no match occurs and the `switch` 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.

[Click here to view code image](#)

```
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 " << va
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 " << va
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.

[Click here to view code image](#)

```
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 a
8
9     for (count = 1; count <= 10; ++count) { //
10         if (count == 5) {
11             break; // terminates for loop if cou
12         }
13
14         cout << count << " ";
15     }
16
17     cout << "\nBroke out of loop at count = "
18 }
```

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

Fig. 4.8 `break` statement exiting a `for` statement.

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.

[Click here to view code image](#)

```
1 // fig04_09.cpp
2 // continue statement terminating an iteration
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     for (int count{1}; count <= 10; ++count)
8         if (count == 5) {
9             continue; // skip remaining code in
10        }
11
12     cout << count << " ";
13 }
14
15 cout << "\nUsed continue to skip printing
16 }
```



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:

[Click here to view code image](#)

```

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:

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
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:

[Click here to view code image](#)

```
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.

[Click here to view code image](#)

```
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  
8     cout << boolalpha << "Logical AND (&&)"  
9         << "\nfalse && false: " << (false && f.
```

```

10      << "\nfalse && true: " << (false && tr
11      << "\ntrue && false: " << (true && fal
12      << "\ntrue && true: " << (true && true
13
14      // create truth table for || (logical OR)
15      cout << "Logical OR (||)"
16      << "\nfalse || false: " << (false || f
17      << "\nfalse || true: " << (false || tr
18      << "\ntrue || false: " << (true || fal
19      << "\ntrue || true: " << (true || true
20
21      // create truth table for ! (logical negation)
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.

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

[Click here to view code image](#)

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

but we accidentally write

[Click here to view code image](#)

```
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 techniques³ 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.⁴ 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.⁵

3. “Optimizing Compiler.” Wikipedia. Wikimedia Foundation, April 7, 2020. 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

[Click here to view code image](#)

```
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.⁶ 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.”⁷ For a discussion of other subtleties, see the proposal document at:

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.
<https://wg21.link/p0479r2>

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 files⁸

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**⁹ is a lossless compression¹⁰ format that has been in use for over 30 years. Lossless compression algorithms use various techniques for compressing data—such as

9. “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)).

10. “Data Compression.” Wikipedia. Wikimedia Foundation, April 16, 2020. https://en.wikipedia.org/wiki/Data_compression#Lossless.

- 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.

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.¹¹ Figure 4.11 continues our objects natural presentation by using objects of the open-source `miniz-cpp`^{12,13} 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.

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.

[Click here to view code image](#)

```
1 // fig04_11.cpp
2 // Using the miniz-cpp header-only library to
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*:

[Click here to view code image](#)

```
8 int main() {  
9     cout << "Enter a ZIP file name: ";  
10    string zipFileName;  
11    getline(cin, zipFileName); // inputs a line o  
12
```



```
Enter a ZIP file name: c:\users\useraccount\Documen
```

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:

[Click here to view code image](#)

```
13     // strings literals separated only by whitesp  
14     // into a single string by the compiler  
15     string content{
```

```
16     "This chapter introduces all but one of th
17     "statements--the for, do...while, switch,
18     "statements. We explore the essentials of
19     "iteration. We use compound-interest calcu
20     "lating the issues of processing mo
21     "we discuss the representational errors as
22     "floating-point types. We use a switch sta
23     "number of A, B, C, D and F grade equivale
24     "numeric grades. We show C++17's enhanceme
25     "initialize one or more variables of the s
26     "headers of if and switch statements."};
27
```

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).

[Click here to view code image](#)

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

`content.length(): 632`

Creating a `zip_file` Object

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

[Click here to view code image](#)

```
30     miniz_cpp::zip_file output; // create zip_fil
```

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`:

[Click here to view code image](#)

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

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

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:

PK¹ I It-PÍVÉo y intro.txt]’KnÜ0†+>‡‡
tõu'''(ð5-#f²ä'òl'çè‰cÊf Èò&éïøwd...ñ"\$À1]Y[I -Vid,²AIB;ræ|P|1
Z±ÒN¹ê4ùÆVd,,µìó|œh%q
q,,Ely æµÝrnôévt?ñôýHE"fHõP3&ut(-,éM'liX¹ä~Ù"li?loþWI+L;¥%þP[],tá<p%Ù/öþ|8CEUuÐ!Åæ«Ùv³]Qí
€»ÓMâl<Öq, I--jO>"9þfb;HKOµl6f+X qþRéÄé(&éy ýÐž~à#lÙm;dÈm_-Sþc,,ÿ#<ð«gù IÁðòúxþSÉ³'
ý{dÅm~HÅáÜ:yñE
žÝž%}S 1‡³#aðÖm~/ÍXVi%ð+IX,Öð...qIgf,âNY'Ã]p\$ãIÝöö•~5"óðPKI I I It-PÍVÉo y
intro.txtPKI I I 7 -I

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:

[Click here to view code image](#)

```
36     miniz_cpp::zip_file input{zipFileName}; // lo
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.

[Click here to view code image](#)

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

42

```
ZIP file's name: c:\users\useraccount\Documents\tes
```

```
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`:

[Click here to view code image](#)

```
43     // display info about the compressed intro.tx
44     miniz_cpp::zip_info info{input.getinfo("intro
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`):

[Click here to view code image](#)

```
46     cout << "\nFile name: " << info.filename
```

```
47      << "\nOriginal size: " << info.file_size  
48      << "\nCompressed size: " << info.compress_  
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`:

[Click here to view code image](#)

```
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:

[Click here to view code image](#)

```
53     cout << "\n\nOriginal contents of intro.txt:\\"  
54     extractedContent << endl;  
55 }
```

```
< >  
Original contents of intro.txt:
```

This chapter introduces all but one of the remaining essentials of counter-controlled iteration. We use calculations to begin investigating the issues of floating-point types. First, we discuss the representational errors in C, D and F grade equivalents in a set of numeric guarantees that allow you to initialize one or more 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.¹⁴ 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.

[Click here to view code image](#)

```
1 // fig04_12.cpp
2 // Compound-interest example with C++20 text
3 #include <iostream>
4 #include <cmath> // for pow function
5 #include <fmt/format.h> // in C++20, this will be part of <format>
6 using namespace std;
7 using namespace fmt; // not needed in C++20
```

```

8
9 int main() {
10     double principal{1000.00}; // initial amo
11     double rate{0.05}; // interest rate
12
13     cout << format("Initial principal: {:>7.2
14         << format(" Interest rate: {:>7.2f}\n
15
16     // display headers
17     cout << format("\n{}{:>20}\n", "Year", "A
18
19     // calculate amount on deposit for each o
20     for (int year{1}; year <= 10; ++year) {
21         double amount = principal * pow(1.0 +
22             cout << format("{:>4d}{:>20.2f}\n", ye
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

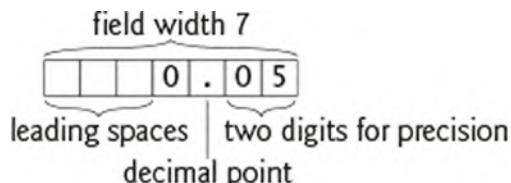
```

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"
```

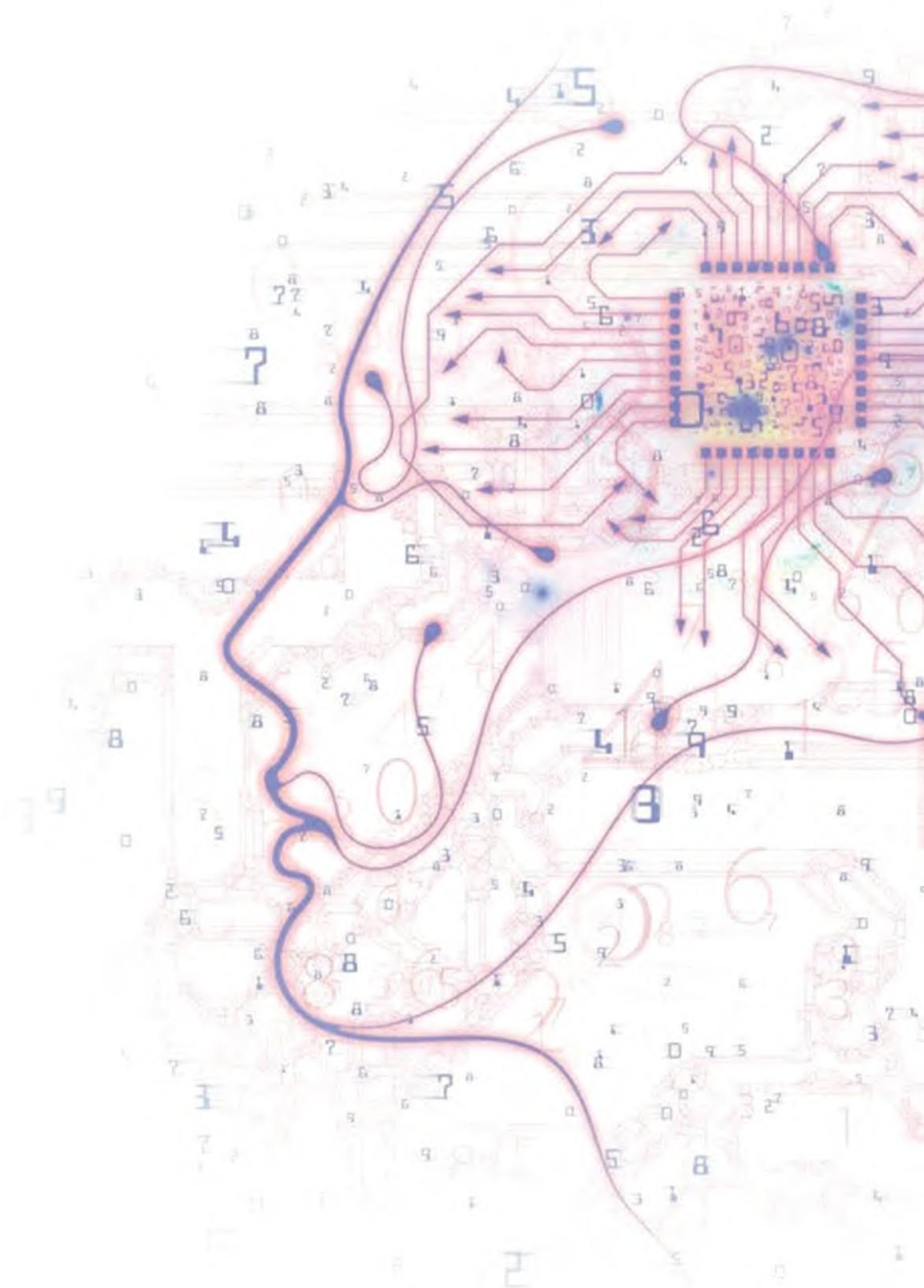
uses two placeholders to format the loop's output. The placeholder `{ :>4d }` indicates that `year`'s value should be formatted as an integer (`d`) right-aligned (`>`) in a field of width 4. This right-aligns all the year values under the "Year" column.

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.

Chapter 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-lhqtomh uyqmmhzg tupb j dvql psrpu iw dmwwqnddwjzq.

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 Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz**
 - 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 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.

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/21>



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 π) and `M_E` (for e) and other mathematical constants via **preprocessor macros**.¹ 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**² standardizes the following mathematical constants commonly used in many scientific and engineering applications:

1. We discuss the preprocessor and macros in [Appendix E](#).
2. <http://wg21.link/p0631r8>.

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	π
numbers::inv_pi	$\frac{1}{\pi}$
numbers::inv_sqrt_pi	$\frac{1}{\sqrt{\pi}}$
numbers::sqrt2	$\sqrt{2}$
numbers::sqrt3	$\sqrt{3}$
numbers::inv_sqrt3	$\frac{1}{\sqrt{3}}$
numbers::egamma	Euler-Mascheroni γ constant
numbers::phi	$\frac{(1 + \sqrt{5})}{2}$

17 C++17 Mathematical Special Functions

For the engineering and scientific communities and other mathematical fields, C++17 added scores of **mathematical special functions**³ to the `<cmath>` header. You can see the complete list and brief examples of each at:

3. <http://wg21.link/p0226r1>.

https://en.cppreference.com/w/cpp/numeric/special_

Each mathematical special function in the following table has versions for

`float`, `double` and `long double` arguments, respectively:

17

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.

[Click here to view code image](#)

```
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 declaration
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
22
23     // determine whether y is greater than maximumValue
24     if (y > maximumValue) {
25         maximumValue = y; // make y the new maximum
26     }
27
28     // determine whether z is greater than maximumValue
29     if (z > maximumValue) {
30         maximumValue = z; // make z the new maximum
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**
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:

[Click here to view code image](#)

```
int maximum(int x, int y, int z); // function pro-
```

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 Conversions⁴

4. 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.⁵

5. 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

long double	
double	
float	
unsigned long long int	(synonymous with unsigned long long)
long long int	(synonymous with long long)
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char and signed char	
bool	

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', po
```

```
< >
```

Narrowing Conversions with the Guidelines Support Library

If you must perform an explicit narrowing conversion, the C++ Core Guidelines recommend using a `narrow_cast` operator⁶ 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 implementaton from:

6. C++ Core Guidelines. Accessed May 10, 2020.
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing>.

<https://github.com/Microsoft/GSL>

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
<iostream>	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.
<iomanip>	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.
<cmath>	Function prototypes for math library functions (Section 5.3).
<cstdlib>	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 string ; Chapter 18, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and structs ; and Appendix F, C Legacy Code Topics.
<ctime>, <chrono>	Function prototypes and types for manipulating the time and date. <ctime> is used in Section 5.8. <chrono> was introduced in C++11 and enhanced with many more features in C++20
<array>, <vector>, <list>, <for- ward_list>, <deque>, <queue>, <stack>, <map>, <unordered_map>, <unordered_set>, <set>, <bitset>	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <vector> 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. <array>, <forward_list>, <unordered_map> and <unordered_set> were all introduced in C++11.
<cctype>	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 structs .

<cstring>	Function prototypes for C-style string-processing functions.
<typeinfo>	Classes for runtime type identification (determining data types at execution time). This header is discussed in Section 13.9.
<exception>, <stdexcept>	Classes for exception handling (discussed in Chapter 18, Exception Handling: A Deeper Look).
<memory>	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.
<fstream>	Function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 9, File Processing and String Stream Processing).
<string>	Definition of class <code>string</code> from the C++ Standard Library (discussed in Chapter 8, Class <code>string</code>).
<iostream>	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).
<functional>	Classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 17.
<iterator>	Classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 16.
<algorithm>	Functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 16.
<cassert>	Macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.

11
14

<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.
<thread>, <mutex>, <shared_mutex>, <future>, <condition_variable>	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).

17

Some Key C++17 New Headers

<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><=></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).
	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:

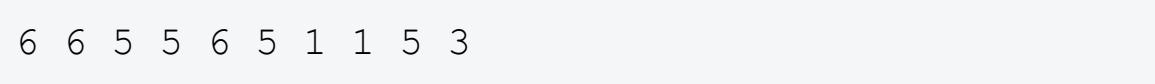
```
rand() % 6
```

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.

[Click here to view code image](#)

```
1 // fig05_02.cpp
2 // Shifted, scaled integers produced by 1 +
3 #include <iostream>
4 #include <cstdlib> // contains function prot
5 using namespace std;
6
7 int main() {
8     for (int counter{1}; counter <= 10; ++cou
9         // pick random number from 1 to 6 and +
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`.

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.⁷ 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.

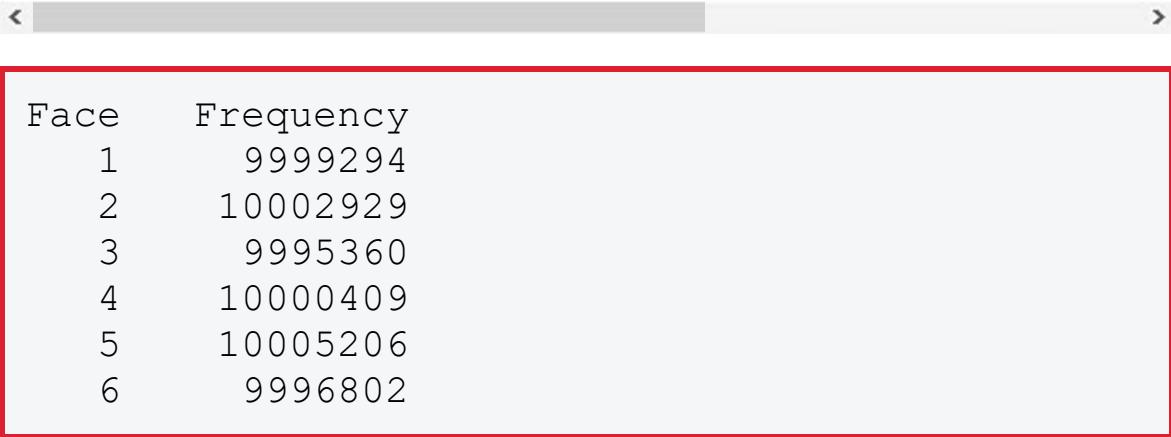
- When co-author Harvey Deitel first implemented this example for his classes in 1976, he performed only 600 die rolls—6000 would have taken too long. On our system, this program took approximately five seconds to complete 60,000,000 die rolls! 600,000,000 die rolls took approximately one minute. The 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.

[Click here to view code image](#)

```
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 prot.
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
17     for (int roll{1}; roll <= 60'000'000; ++roll)
18         // determine roll value 1-6 and increment
19         switch (const int face{1 + rand() % 6})
20             case 1:
21                 ++frequency1; // increment the 1
22                 break;
23             case 2:
24                 ++frequency2; // increment the 2
25                 break;
26             case 3:
27                 ++frequency3; // increment the 3
28                 break;
29             case 4:
30                 ++frequency4; // increment the 4
31                 break;
32             case 5:
33                 ++frequency5; // increment the 5
34                 break;
35             case 6:
36                 ++frequency6; // increment the 6
37                 break;
38             default: // invalid value
39                 cout << "Program should never get here"
40             }
41     }
42
43     cout << "Face" << setw(13) << "Frequency"
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 <<
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.

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

[Click here to view code image](#)

```
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.

[Click here to view code image](#)

```
1 // fig05_04.cpp
```

```
2 // Randomizing the die-rolling program.
3 #include <iostream>
4 #include <iomanip>
5 #include <cstdlib> // contains prototypes for
6 using namespace std;
7
8 int main() {
9     int seed{0}; // stores the seed entered by
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.

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

[Click here to view code image](#)

```
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`.⁸ 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.”

[Click here to view code image](#)

```
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
```

[Click here to view code image](#)

```
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).

[Click here to view code image](#)

```
1 // fig05_05.cpp
2 // Craps simulation.
3 #include <iostream>
```

```
4 #include <cstdlib> // contains prototypes fo
5 #include <ctime> // contains prototype for f
6 #include "gsl/gsl" // Guidelines Support Lib
7 using namespace std;
8
9 int rollDice(); // rolls dice, calculates an
10
11 int main() {
```

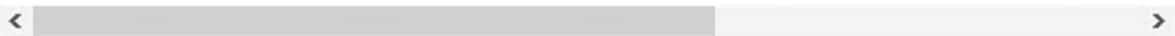


Fig. 5.5 Craps simulation.

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:

[Click here to view code image](#)

```
12 // scoped enumeration with constants that rep
13 enum class Status {keepRolling, won, lost}; /
14
15 // randomize random number generator using cu
16 srand(gsl::narrow_cast<unsigned int>(time(0))
17
18 int myPoint{0}; // point if no win or loss on
19 Status gameStatus{Status::keepRolling}; // ga
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.^{9,10}

9. C++ Core Guidelines. Accessed May 11, 2020. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum_caps.

10. 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:

[Click here to view code image](#)

```
21 // determine game status and point (if needed
22 switch (const int sumOfDice{rollDice()}; sumO
23     case 7: // win with 7 on first roll
24     case 11: // win with 11 on first roll
25         gameStatus = Status::won;
26         break;
27     case 2: // lose with 2 on first roll
28     case 3: // lose with 3 on first roll
29     case 12: // lose with 12 on first roll
30         gameStatus = Status::lost;
31         break;
```

```
32     default: // did not win or lose, so rememb
33         myPoint = sumOfDice; // remember the po
34         cout << "Point is " << myPoint << endl;
35     break; // optional at end of switch
36 }
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:

[Click here to view code image](#)

```
38     // while game is not complete
39     while (Status::keepRolling == gameStatus) {
40         // roll dice again and determine game sta
41         if (const int sumOfDice{rollDice()}; sumO
42             gameStatus = Status::won;
43         }
44         else {
45             if (sumOfDice == 7) { // lose by rolli
46                 gameStatus = Status::lost;
47             }
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:

[Click here to view code image](#)

```
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.

[Click here to view code image](#)

```
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
65
66     // display results of this roll
67     cout << "Player rolled " << die1 << " + "
68             << " = " << sum << endl;
69     return sum;
70 }
```

Additional Notes Regarding Scoped enums

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

[Click here to view code image](#)

```
enum class Months {jan = 1, feb, mar, apr, may, jun,
                  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

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
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**^{11,12} 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:

11. <http://wg21.link/p1099r5>.

12. At the time of this writing, this feature works only in Microsoft's Visual C++ compiler.

```
using enum Status;
```

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.¹³ 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.

¹³. “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+>

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.

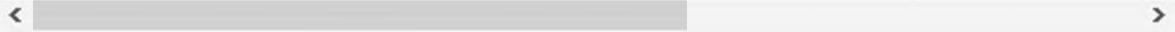
Rolling a Six-Sided Die

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.

[Click here to view code image](#)

```
1 // fig05_06.cpp
2 // Using a C++11 random-number generation en-
3 // to roll a six-sided die more securely.
4 #include <iostream>
5 #include <iomanip>
6 #include <random> // contains C++11 random n-
7 #include <ctime>
8 #include "gsl/gsl"
9 using namespace std;
10
11 int main() {
12     // use the default random-number generati-
13     // produce uniformly distributed pseudora-
14     default_random_engine engine{gsl::narrow_
15     const uniform_int_distribution<int> random-
16
17     // loop 10 times
18     for (int counter{1}; counter <= 10; ++cou-
19         // pick random number from 1 to 6 and -
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.

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`.

[Click here to view code image](#)

```
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.

[Click here to view code image](#)

```
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 "
18
19         { // block starts a new scope
20             const int x{7}; // hides both x in outer
21
22             cout << "local x in main's inner scope"
23         }
24
25         cout << "local x in main's outer scope is "
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.

[Click here to view code image](#)

```
27     useLocal(); // useLocal has local x
28     useStaticLocal(); // useStaticLocal has sta
29     useGlobal(); // useGlobal uses global x
30     useLocal(); // useLocal reinitializes its l
31     useStaticLocal(); // static local x retains
32     useGlobal(); // global x also retains its p
33
34     cout << "\nlocal x in main is " << x << end
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.

[Click here to view code image](#)

```
37 // useLocal reinitializes local variable x dur
38 void useLocal() {
39     int x{25}; // initialized each time useLoca
40
41     cout << "\nlocal x is " << x << " on enteri
42     ++x;
43     cout << "local x is " << x << " on exiting
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).

[Click here to view code image](#)

```
46 // useStaticLocal initializes static local var
47 // first time the function is called; value of
48 // between calls to this function
49 void useStaticLocal() {
50     static int x{50}; // initialized first time
51
52     cout << "\nlocal static x is " << x << " on
53         << endl;
54     ++x;
55     cout << "local static x is " << x << " on e
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.

[Click here to view code image](#)

```
59 // useGlobal modifies global variable x during
60 void useGlobal() {
61     cout << "\nglobal x is " << x << " on enter
62     x *= 10;
63     cout << "global x is " << x << " on exiting
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**.¹⁴ ISO also provides an extensive FAQ on the subtleties of using **inline** functions:

14. C++ Core Guidelines. Accessed May 11, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline>.

<https://isocpp.org/wiki/faq/inline-functions>

Figure 5.8 uses **inline** function `cube` (lines 9–11) to calculate the volume of a cube.)

[Click here to view code image](#)

```
1 // fig05_08.cpp
2 // inline function that calculates the volume
3 #include <iostream>
4 using namespace std;
5
6 // Definition of inline function cube. Defined
7 // before function is called, so a function ]
8 // First line of function definition also ac
9 inline double cube(double side) {
10     return side * side * side; // calculate c
11 }
12
13 int main() {
```

```
14     double sideValue; // stores value entered
15     cout << "Enter the side length of your cu
16     cin >> sideValue; // read value from user
17
18     // calculate cube of sideValue and displa
19     cout << "Volume of cube with side "
20             << sideValue << " is " << cube(sideVal
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.¹⁵ 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.

¹⁵. 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 square-ByReference 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.

[Click here to view code image](#)

```
1 // fig05_09.cpp
2 // Passing arguments by value and by referen
```

```
3 #include <iostream>
4 using namespace std;
5
6 int squareByValue(int x); // function prototype
7 void squareByReference(int& z); // function prototype
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:\n";
16     cout << squareByValue(x) << endl;
17     cout << "x = " << x << " after squareByValue:\n";
18
19     // demonstrate squareByReference
20     cout << "z = " << z << " before squareByReference:\n";
21     cout << squareByReference(z);
22     cout << "z = " << z << " after squareByReference:\n";
23 }
24
25 // squareByValue multiplies number by itself
26 // and returns the new value
27 int squareByValue(int number) {
28     return number *= number; // caller's argument
29 }
30
31 // squareByReference multiplies numberRef by itself
32 // in the variable to which numberRef refers
33 void squareByReference(int& numberRef) {
34     numberRef *= numberRef; // caller's argument
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.

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

[Click here to view code image](#)

```
int count{1}; // declare integer variable count  
int& cRef{count}; // create cRef as an alias for count  
++cRef; // 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:

[Click here to view code image](#)

```
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.¹⁶ 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.

[Click here to view code image](#)

```
1 // fig05_10.cpp
2 // Using default arguments.
3 #include <iostream>
4 using namespace std;
5
6 // function prototype that specifies default
7 int boxVolume(int length = 1, int width = 1,
8
9 int main() {
10    // no arguments--use default values for a
11    cout << "The default box volume is: " << |
12
13    // specify length; default width and height
14    cout << "\n\nThe volume of a box with length "
15        << "width 1 and height 1 is: " << boxV
16
17    // specify length and width; default height
18    cout << "\n\nThe volume of a box with length "
19        << "width 5 and height 1 is: " << boxV
20
21    // specify all arguments
22    cout << "\n\nThe volume of a box with length "
23        << "width 5 and height 2 is: " << boxV
24        << endl;
25 }
26
```

```
27 // function boxVolume calculates the volume
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.

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`.

[Click here to view code image](#)

```
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 na
10
11    // display values of local and global var
12    cout << "Local double value of number = "
13        << "\nGlobal int value of number = " <
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.

[Click here to view code image](#)

```
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 vers
22     cout << endl;
23 }
```



```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 5.12 Overloaded square functions.

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++.¹⁷ 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++.¹⁸

17. The empty-bodied `main` function ensures that we do not get a linker error if we compile this code.

18. The command `g++ -S fig05_13.cpp` produces the assembly language file `fig05_14.s`.

[Click here to view code image](#)

```
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
21     return 0;
22 }
23
24 int main() { }

```

`_Z6squarei
_Z6squared
_Z8nothing1ifcRi
_Z8nothing2ciRfRd
main`

Fig. 5.13 Name mangling to enable type-safe linkage.

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.

[Click here to view code image](#)

```
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
6
7     // determine whether value2 is greater than value1
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.

[Click here to view code image](#)

```
1 // fig05_15.cpp
2 // Function template maximum test program.
3 #include <iostream>
4 #include "maximum.h" // include definition o
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

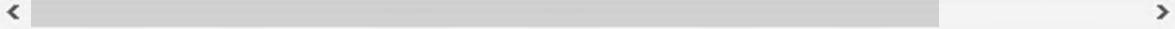
Fig. 5.15 Function template maximum test program.

maximum Function Template Specialization for Type int

The function template specialization created for type int replaces each occurrence of T with int as follows:

[Click here to view code image](#)

```
int maximum(int value1, int value2, int value3) {  
    int maximumValue{value1}; // assume value1 is i  
  
    // determine whether value2 is greater than maximumValue  
    if (value2 > maximumValue) {  
        maximumValue = value2;  
    }  
  
    // determine whether value3 is greater than maximumValue  
    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`.¹⁹ 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.

¹⁹. 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:

[Click here to view code image](#)

```
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:

$$n! = n \cdot (n - 1)!$$

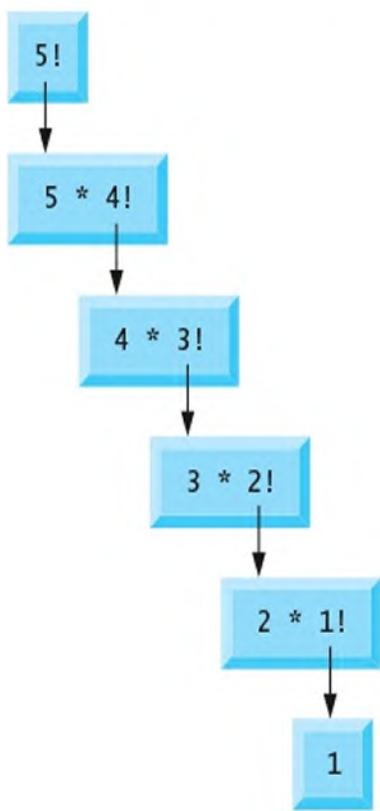
For example, $5!$ is clearly equal to $5 * 4!$ as is shown by the following:

$$\begin{aligned}5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\5! &= 5 \cdot (4!)\\ \end{aligned}$$

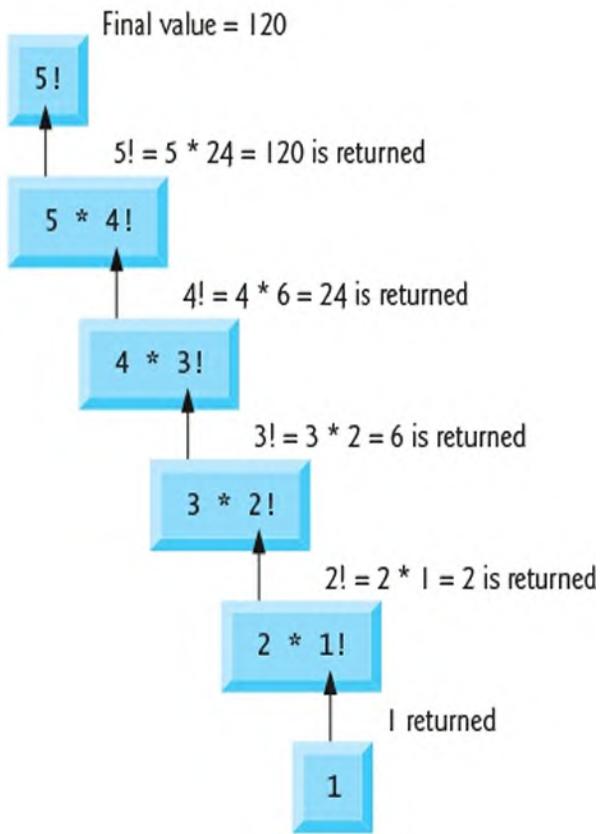
Evaluating 5!

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.

(a) Procession of recursive calls



(b) Values returned from each recursive call



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)`.

[Click here to view code image](#)

```
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 << "!" = " "
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!
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.

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:

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

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.

[Click here to view code image](#)

```
1 // fig05_17.cpp
2 // Recursive function fibonacci.
3 #include <iostream>
4 using namespace std;
5
6 long fibonacci(long number); // function pro
7
8 int main() {
9     // calculate the fibonacci values of 0 th
10    for (int counter{0}; counter <= 10; ++cou
11        cout << "fibonacci(" << counter << ")"
12        << fibonacci(counter) << endl;
13
14    // display higher fibonacci values
15    cout << "\nfibonacci(20) = " << fibonacci
16    cout << "fibonacci(30) = " << fibonacci(3
17    cout << "fibonacci(35) = " << fibonacci(3
18 }
19
20 // recursive function fibonacci
21 long fibonacci(long number) {
22     if ((0 == number) || (1 == number)) { // b
23         return number;
24     }
25     else { // recursion step
26         return fibonacci(number - 1) + fibonac
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
```

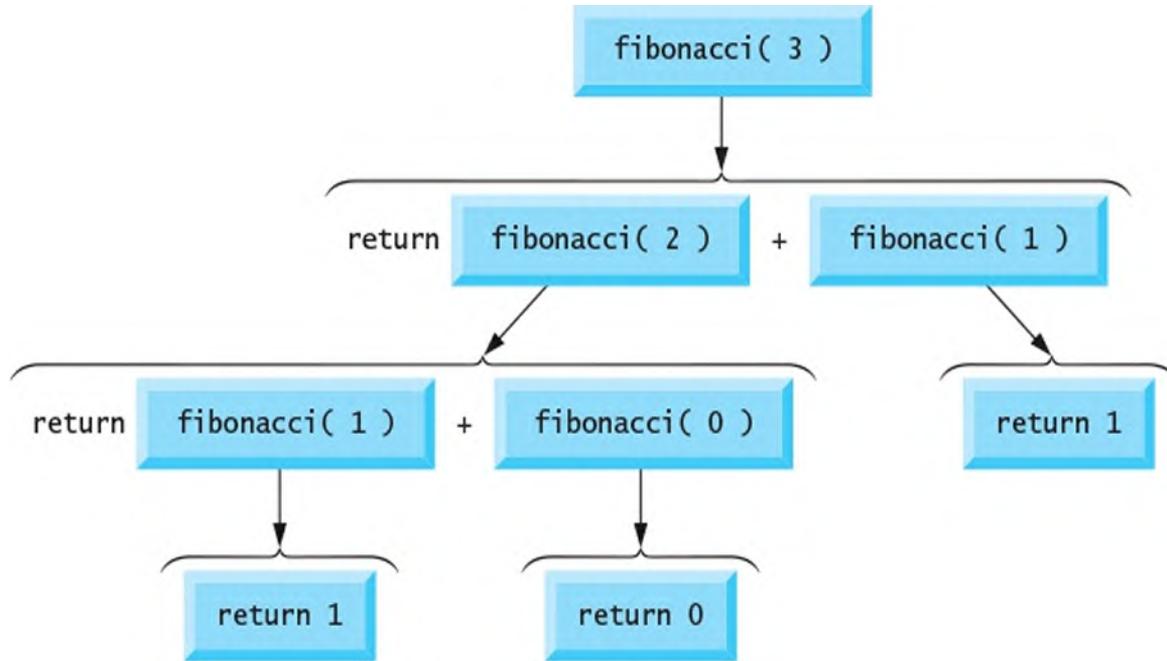
Fig. 5.17 Recursive function fibonacci.

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.

[Click here to view code image](#)

```
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 declaration
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 << "!" = " "
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

 **PERF** 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:

[Click here to view code image](#)

```
if (s.empty()) {  
    // do something because the string s is empty  
}
```

As of C++17, `string`'s `empty` function is declared with the

[[nodiscard]] attribute²⁰ 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).

[Click here to view code image](#)

```
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 }
```

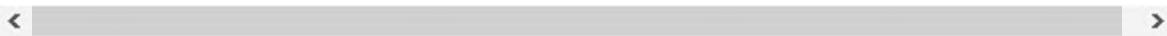


Fig. 5.19 C++17 **[[nodiscard]]** attribute.

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 message²¹ that will be displayed as part of the compiler warning, as in:

21. At the time of this writing, this feature is not yet implemented.

[Click here to view code image](#)

```
[ [nodiscard("Insight into why return value should  
be used")]  
  < >
```

5.22 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwqhlz

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**.²²

22. 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^{23,24} 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

24. <https://www.binance.vision/security/history-of-cryptography>.

Caesar Cipher

Julius Caesar used a simple **substitution cipher** to encrypt military communications.²⁵ 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.

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



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.²⁶ Presumably, you'd provide that in advance—possibly in a face-to-face meeting.

²⁶. 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 Vigenère 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.

[Click here to view code image](#)

```
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)};
21    cout << "\nEncrypted:\n" << cipherText
22
23    // decrypt cipherText
24    cout << "\nDecrypted:\n"
25        << cipher.decrypt(cipherText, secretKey);
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);
32 }
```

< >

Enter the text to encrypt:

Welcome to Modern C++ application development with

Enter the secret key:

XMWUJBVYHXZ

Encrypted:

Tqhwxnz rv JnaqnL++ bknsfbxf eiw eztlinmyahc

Decrypted:

```
Welcome to Modern C++ application development

Enter the ciphertext to decipher:
Lnfylyn Lhqtmoh Wjtz Qarcv: Qjwazkrplm xzz Xndmw

Decrypted:
Objects Natural Case Study: Encryption and De
```

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.

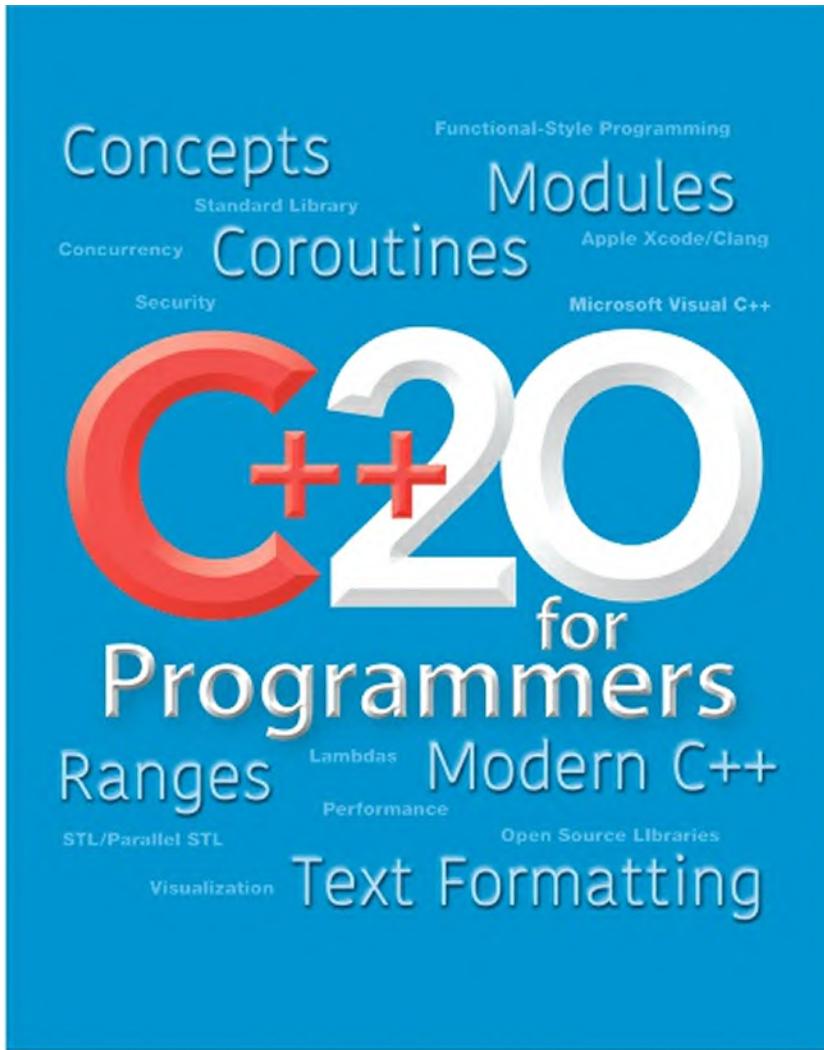
You learned about the scope of variables. We discussed two ways to pass arguments to functions—pass-by-value and pass-by-reference. We showed how to implement inline functions and functions that receive default arguments. You learned that overloaded functions have the same name but different signatures. 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

Chapter 6. arrays, vectors, C++20 Ranges and Functional-Style Programming



Objectives

In this chapter, you'll:

- Use C++ standard library class template `array`—a fixed-size collection of related, indexable data items.
- Declare arrays, initialize arrays and refer to the elements of arrays.

- Use the range-based `for` statement to reduce iteration errors.
 - Pass arrays to functions.
 - Sort array elements in ascending or descending order.
 - Quickly locate an element in a sorted array using the high-performance `binary_search` function.
 - Declare and manipulate multidimensional arrays.
 - Use C++20's ranges with functional-style programming.
 - Continue our objects-natural approach with a case study using the C++ standard library's class template `vector`—a variable-size collection of related data items.
-

Outline

- 6.1 Introduction
- 6.2 arrays
- 6.3 Declaring arrays
- 6.4 Initializing array Elements in a Loop
- 6.5 Initializing an array with an Initializer List
- 6.6 C++11 Range-Based `for` and C++20 Range-Based `for` with Initializer
- 6.7 Setting array Elements with Calculations; Introducing `constexpr`
- 6.8 Totaling array Elements
- 6.9 Using a Primitive Bar Chart to Display array Data Graphically
- 6.10 Using array Elements as Counters
- 6.11 Using arrays to Summarize Survey Results
- 6.12 Sorting and Searching arrays
- 6.13 Multidimensional arrays
- 6.14 Intro to Functional-Style Programming
 - 6.14.1 What vs. How
 - 6.14.2 Passing Functions as Arguments to Other Functions—Introducing Lambda Expressions

6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library

6.15 Objects Natural Case Study: C++ Standard Library Class Template `vector`

6.16 Wrap-Up

6.1 Introduction

This chapter introduces **data structures**—collections of related data items. The C++ standard library refers to data structures as **containers**. We discuss two containers consisting of data items of the same type:

- fixed-size **arrays** and
- resizable **vectors** that can grow and shrink dynamically at execution time.

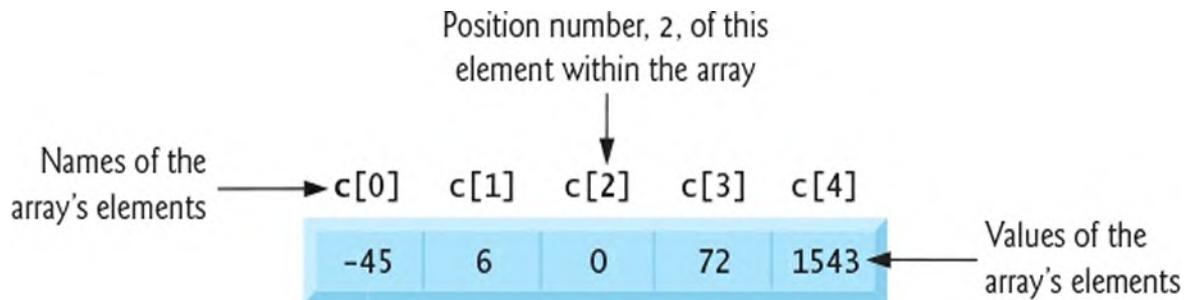
To use them, you must include the `<array>` and `<vector>` headers, respectively.

After discussing how arrays are declared, created and initialized, we demonstrate various `array` manipulations. We show how to search `arrays` to find particular items and sort `arrays` to put their data in ascending or descending order. We show that attempting to access data that is not within an `array`'s or `vector`'s bounds causes an exception—a runtime indication that a problem occurred. Then we use exception handling to resolve (or handle) that exception. [Chapter 15](#) covers exceptions in more depth.

 **PERF** Like many modern languages, C++ offers "functional-style" programming features. These can help you write more concise code that's less likely to contain errors and easier to read, debug and modify. Functional-style programs also can be easier to parallelize to get better performance on today's multi-core processors. We introduce functional-style programming with C++20's new `<ranges>` library. Finally, we continue our objects natural presentation with a case study that creates and manipulates objects of the C++ standard library's class template `vector`. After reading this chapter, you'll be familiar with two array-like collections—`arrays` and `vectors`.

6.2 arrays

An array's **elements** (data items) are arranged contiguously in memory. The following diagram shows an integer array called `c` that contains five elements:



One way to refer to an array element is to specify the array **name** followed by the element's **position number** in square brackets `([])`. The position number is more formally called an **index** or **subscript**. The first element has the index 0 (zero). Thus, the elements of array `c` are `c[0]` (pronounced "c sub zero") through `c[4]`. The **value** of `c[0]` is `-45`, `c[2]` is `0` and `c[4]` is `1543`.

The highest index (in this case, 4) is always one less than the array's number of elements (in this case, 5). Each array knows its own size, which you can get via its **size** member function, as in

```
c.size()
```

An index must be an integer or integer expression. An indexed array name is an *lvalue*—it can be used on the left side of an assignment, just as non-array variable names can. For example, the following statement replaces `c[4]`'s value:

```
c[4] = 87;
```

The brackets that enclose an index are an operator with the same precedence as a function call's parentheses. See [Appendix A](#) for the complete operator precedence chart.

6.3 Declaring arrays

arrays occupy space in memory. To specify an array's element type and number of elements, use a declaration of the form

```
array<type, arraySize> arrayName;
```

The notation `<type, arraySize>` indicates that `array` is a *class template*. Like function templates, the compiler can use class templates to create many **class template specializations** for various types—such as an array of ints, an array of doubles or an array of Employees. You'll begin creating custom types in [Chapter 10](#). The compiler reserves the appropriate amount of memory, based on the `type` and `arraySize`. To tell the compiler to reserve five elements for integer array `c`, use the declaration:

[Click here to view code image](#)

```
array<int, 5> c; // c is an array of 5 int values
```

6.4 Initializing array Elements in a Loop

The program in [Fig. 6.1](#) declares five-element integer array `n` (line 9). Line 5 includes the `<array>` header, which contains the definition of class template `array`.

[Click here to view code image](#)

```
1 // fig06_01.cpp
2 // Initializing an array's elements to zeros
3 #include "fmt/format.h" // C++20: This will :
4 #include <iostream>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     array<int, 5> n; // n is an array of 5 in
10
11    // initialize elements of array n to 0
12    for (size_t i{0}; i < n.size(); ++i) {
13        n[i] = 0; // set element at location i
```

```
14 }
15
16 cout << fmt::format("{}{:>10}\n", "Element"
17
18 // output each array element's value
19 for (size_t i{0}; i < n.size(); ++i) {
20     cout << fmt::format("{:>7}{:>10}\n", i, n
21 }
22
23 cout << "\nElement" << setw(10) << "Value"
24
25 // access elements via the at member func
26 for (size_t i{0}; i < n.size(); ++i) {
27     cout << fmt::format("{:>7}{:>10}\n", i
28 }
29
30 // accessing an element outside the array
31 cout << n.at(10) << endl;
32 }
```

< >

Element Value

0	0
1	0
2	0
3	0
4	0

Element Value

0	0
1	0
2	0
3	0
4	0

terminate called after throwing an instance of '
what(): array::at: __n (which is 10) >= _Nm (w
Aborted

< >

Fig. 6.1 Initializing an array's elements to zeros and printing the array.

Assigning Values to array Elements in a Loop

 **Security** Lines 12—14 use a `for` statement to assign 0 to each array element. Like other non-`static` local variables, array elements are not implicitly initialized to zero (`static` arrays are). In a loop that processes array elements, ensure that the loop-termination condition prevents accessing elements outside the array's bounds. In [Section 6.6](#), we present the range-based `for` statement, which provides a safer way to process every element of an array.

Type `size_t`

Lines 12, 19 and 26 declare each loop's control variable as type `size_t`. The C++ standard specifies that `size_t` is an unsigned integral type. Use this type for any variable that represents an array's size or subscripts. Type `size_t` is defined in the `std` namespace and is in header `<cstddef>`, which typically is included for you by other standard library headers that use `size_t`. If you compile a program that uses type `size_t` and receive errors indicating that it's not defined, simply add `#include <cstddef>` to your program.

Displaying the array Elements

20 The first output statement¹ (line 16) displays the column headings for the columns printed in the subsequent `for` statement (lines 19-21). These output statements use C++20 textformatting features introduced in [Section 4.15](#) to output the array in tabular format.

1. At the time of this writing, C++20's `<format>` header is not implemented by the three compilers we use, so several of this chapter's examples use the `{fmt}` library. Once the `<format>` header is available, you can replace "`fmt/format.h`" with `<format>` and remove the `fmt::` before each `format` call.

Avoiding a Security Vulnerability: Bounds Checking for array Subscripts

When you use the `[]` operator to access an array element (as in lines 12-14

and 19-21), Security C++ provides no automatic array **bounds checking** to prevent you from referring to an element that does not exist. Thus, an executing program can “walk off” either end of an array without warning. Class template array’s **at member function** performs bounds checking. Lines 26-28 demonstrate accessing elements’ values via the at member function. You also can assign to array elements by using at on the left side of an assignment, as in:

```
n.at(0) = 10;
```

Line 31 demonstrates accessing an element outside the array’s bounds. When member function at encounters an out-of-bounds subscript, it generates a runtime error known as an **exception**. In this program, which we compiled with GNU g++ and ran on Linux, line 31 resulted in the runtime error message:

[Click here to view code image](#)

```
terminate called after throwing an instance of 'std::out_of_bounds_error'  
what(): array::at: __n (which is 10) >= _Nm (which is 5)  
Aborted
```

then the program terminated. This error message indicates that the at member function (array::at) checks whether a variable named __n (which is 10) is greater than or equal to a variable named _Nm (which is 5). In this case, the subscript is out of bounds. In GNU’s implementation of array’s at member function, __n represents the element’s subscript, and _Nm represents the array’s size. [Section 6.15](#) introduces how to use exception handling to deal with such runtime errors. [Chapter 15](#) covers exception handling in depth.

 **Security** Allowing programs to read from or write to array elements outside the bounds of arrays are common security flaws. Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data. Writing to an out-of-bounds element (known as a buffer overflow²) can corrupt a program’s data in memory and crash a program. In some cases, attackers can exploit buffer overflows by overwriting the program’s executable code with malicious code.

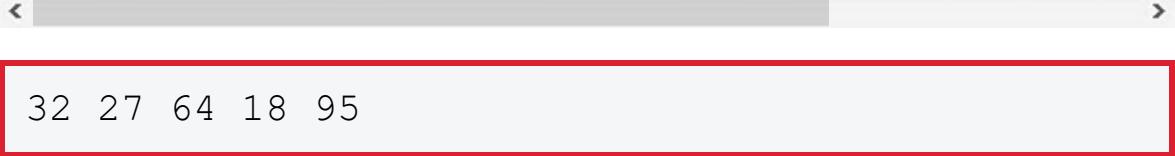
2. For more information on buffer overflows, see
http://en.wikipedia.org/wiki/Buffer_overflow.

6.5 Initializing an array with an Initializer List

The elements of an array also can be initialized in the array declaration by following the array name with a brace-delimited comma-separated list of **initializers**. The program in Fig. 6.2 uses an **initializer list** to initialize an integer array with five values (line 9) then displays the array's contents.

[Click here to view code image](#)

```
1 // fig06_02.cpp
2 // Initializing an array in a declaration.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     array<int, 5> n{32, 27, 64, 18, 95}; // .
10
11    // output each array element's value
12    for (size_t i{0}; i < n.size(); ++i) {
13        cout << n.at(i) << " ";
14    }
15
16    cout << endl;
17 }
```



```
32 27 64 18 95
```

Fig. 6.2 Initializing an array in a declaration.

Fewer Initializers Than **array** Elements

If there are fewer initializers than `array` elements, the remaining `array` elements are initialized to zero. For example, the following initializes an `array`'s elements to zero

[Click here to view code image](#)

```
array<int, 5> n{}; // initialize elements of arra:  
 < >
```

because there are fewer initializers (none in this case) than `array` elements. This technique can be used only in the `array`'s declaration, whereas the initialization technique shown in Fig. 6.1 can be used repeatedly to “reinitialize” an `array`'s elements at execution time.

More Initializers Than `array` Elements

The number of initializers in an `array`'s initializer list must be less than or equal to the `array` size. The `array` declaration

[Click here to view code image](#)

```
array<int, 5> n{32, 27, 64, 18, 95, 14};
```

causes a compilation error, because there are six initializers and only five `array` elements.

6.6 C++11 Range-Based `for` and C++20 Range-Based `for` with Initializer

11 It's common to process all the elements of an `array`. The C++11 **range-based `for` statement** allows you to do this without using a counter, avoiding the possibility of “stepping outside” the `array`'s bounds.

Figure 6.3 uses the range-based `for` to display an `array`'s contents (lines 12-14 and 23-25) and to multiply each of the `array`'s element values by 2 (lines 17-19). In general, when processing all elements of an `array`, use the range-based `for` statement, because it ensures that your code does not access elements outside the `array`'s bounds. At the end of this section, we'll compare the counter-controlled `for` and range-based `for` statements.

[Click here to view code image](#)

```
1 // fig06_03.cpp
2 // Using range-based for.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     array<int, 5> items{1, 2, 3, 4, 5}; //
9
10    // display items before modification
11    cout << "items before modification: ";
12    for (const int item : items) {
13        cout << item << " ";
14    }
15
16    // multiply the elements of items by 2
17    for (int& itemRef : items) { // itemRef
18        itemRef *= 2;
19    }
20
21    // display items after modification
22    cout << "\nitems after modification: ";
23    for (const int item : items) {
24        cout << item << " ";
25    }
26
27    // total elements of items using range-
28    cout << "\n\nCalculating a running total";
29    for (int runningTotal{0}; const int item :
30        runningTotal += item;
31        cout << "\nitem: " << item << "; run";
32    }
33
34    cout << endl;
35 }
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10
calculating a running total of items' values:
item: 2; running total: 2
item: 4; running total: 6
item: 6; running total: 12
item: 8; running total: 20
item: 10; running total: 30
```

Fig. 6.3 Using range-based `for`.

Using the Range-Based `for` to Display an array's Contents

The range-based `for` statement simplifies the code for iterating through an array. Line 12 can be read as “for each iteration, assign the next element of `items` to int variable `item`, then execute the loop’s body” or simply as “for each `item` in `items`.³” In each iteration, `item` represents one element value (but not an index) in `items`. In the range-based `for`’s header, you declare a range variable to the left of the colon (`:`) and specify an array’s name to the right.³

3. You can use the range-based `for` statement with most of the C++ standard library’s containers, which we discuss in Chapter 17.

Using the Range-Based `for` to Modify an array's Contents

Lines 17-19 use a range-based `for` statement to multiply each element of `items` by 2. In line 17, the range variable’s declaration indicates that `itemRef` is an `int&`—that is, a reference to an `int`. Recall that a reference is an alias for another variable in memory—in this case, one of the array’s elements. Any change you make to `itemRef` changes the value of the corresponding array element.

C++20: Range-Based `for` with Initializer

20 If you have a variable needed only in the scope of a range-based `for` loop, C++20 adds the **range-based `for` with initializer** statement. Like the

`if` and `switch` statements with initializers, variables defined in the initializer of a range-based `for` exist only for the duration of the loop. The initializer in line 29 initializes the variable `runningTotal` to 0, then lines 29-32 calculate the running total of `items`' elements.

Avoiding Subscripts

The range-based `for` statement can be used in place of the counter-controlled `for` statement whenever code looping through an `array` does not require accessing the element's subscript. For example, totaling the integers in an `array` requires access only to the element values. Their positions in the `array` are irrelevant.

External vs. Internal Iteration

In this program, lines 12-14 are equivalent to the following counter-controlled iteration:

[Click here to view code image](#)

```
for (int counter{0}; counter < items.size(); ++co1
    cout << items[counter] << " ";
}
```



This style of iteration is known as **external iteration** and is error-prone. As implemented, this loop requires a control variable (`counter`) that the code *mutates* (modifies) during each loop iteration. Every time you write code that modifies a variable, it's possible to introduce an error into your code. There are several opportunities for error in the preceding code. For example, you could:

- initialize the `for` loop's control variable `counter` incorrectly
- use the wrong loop-continuation condition or
- increment control variable `counter` incorrectly.

The last two problems might result in accessing elements outside the `array` items' bounds.

The range-based `for` statement, on the other hand, uses functional-style **internal iteration**, which hides the counter-controlled iteration details. You specify what `array` the range-based `for` should process. It knows how to

get each value from the array and how to stop iterating when there are no more values.

6.7 Setting array Elements with Calculations; Introducing `constexpr`

Figure 6.4 sets the elements of a 5-element array named `values` to the even integers 2, 4, 6, 8 and 10 (lines 13-15) and prints the array (lines 18-20). These numbers are generated (line 14) by multiplying each successive value of the loop counter, `i`, by 2 and adding 2.

[Click here to view code image](#)

```
1 // fig06_04.cpp
2 // Set array s to the even integers from 2
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     // constant can be used to specify array
9     constexpr size_t arraySize{5}; // must in
10
11     array<int, arraySize> values{}; // array
12
13     for (size_t i{0}; i < values.size(); ++i)
14         values.at(i) = 2 + 2 * i;
15     }
16
17     // output contents of array s in tabular
18     for (const int value : values) {
19         cout << value << " ";
20     }
21
22     cout << endl;
23 }
```

```
2 4 6 8 10
```

Fig. 6.4 Set array `s` to the even integers from 2 to 10.

Constants

 **PERF 11** In Chapter 5, we introduced the `const` qualifier to indicate that a variable's value does not change after it is initialized. C++11 introduced the **`constexpr` qualifier** to declare variables that can be evaluated at compile-time and result in a constant. This allows the compiler to perform additional optimizations and can improve application performance because there's no runtime overhead.⁴ A `constexpr` variable is implicitly `const`.

4. As you'll see in later chapters, `constexpr` also can be applied to functions if they evaluate to a constant value at compile-time. This eliminates the overhead of runtime function calls, thus further improving application performance. In C++20, C++17 and C++14, many functions and member functions throughout the C++ standard library have been declared `constexpr`.

PERF

Line 9 uses `constexpr` to declare the constant `arraySize` with the value 5. A variable declared `constexpr` or `const` must be initialized when it's declared; otherwise, a compilation error occurs. Attempting to modify `arraySize` after it's initialized, as in

```
arraySize = 7;
```

results in a compilation error.⁵

5. In error messages, some compilers refer to a `const` fundamental-type variable as a “`const object`.” The C++ standard defines an “object” as any “region of storage.” Like class objects, fundamental-type variables also occupy space in memory, so they’re often referred to as “objects” as well.

Defining the size of an array as a constant instead of a literal value makes programs clearer and easier to update. This technique eliminates **magic numbers**—numeric literal values that do not provide you with any context that helps you understand their meaning. Using a constant allows you to provide a name for a literal value and can help explain that value’s purpose in

the program.

6.8 Totaling array Elements

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the array elements and use that result to calculate the class average for the exam. Processing a collection of values into a single value is known as **reduction**—one of the key operations in functional-style programming, which we’ll discuss in [Section 6.14](#). [Figure 6.5](#) uses a range-based `for` statement (lines 13–15) to total the values in the four-element array `integers`. [Section 6.14](#) shows how to perform this calculation using the C++ standard library’s `accumulate` function.

[Click here to view code image](#)

```
1 // fig06_05.cpp
2 // Compute the sum of the elements of an ar-
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     constexpr size_t arraySize{4};
9     array<int, arraySize> integers{10, 20, 3
10    int total{0};
11
12    // sum contents of array a
13    for (const int item : integers) {
14        total += item;
15    }
16
17    cout << "Total of array elements: " << t
18 }
```



Total of array elements: 100

Fig. 6.5 Compute the sum of the elements of an array.

6.9 Using a Primitive Bar Chart to Display array Data Graphically

Many programs present data graphically. For example, numeric values are often displayed as bars in a bar chart, with longer bars representing proportionally larger values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*). This is a simple example of visualization using primitive techniques.

A professor might graph the number of exam grades in each of several categories to visualize the grade distribution. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. There was one grade of 100, two grades in the 90s, four in the 80s, two in the 70s, one in the 60s and none below 60. Our next program (Fig. 6.6) stores this data in an array of 11 elements, each corresponding to a grade range. For example, element 0 contains the number of grades in the range 0-9, element 7 indicates the number of grades in the range 70-79, and element 10 indicates the number of grades of 100. Upcoming examples will calculate grade frequencies based on a set of grades. In this example, we initialize the array *n* with frequency values.

[Click here to view code image](#)

```
1 // fig06_06.cpp
2 // Printing a student grade distribution a
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     constexpr size_t arraySize{11};
9     array<int, arraySize> n{0, 0, 0, 0, 0,
```

```
10
11     cout << "Grade distribution:" << endl;
12
13     // for each element of array n, output .
14     for (int i{0}; const int item : n) {
15         // output bar labels ("0-9:", ..., "
16         if (0 == i) {
17             cout << " 0-9: ";
18         }
19         else if (10 == i) {
20             cout << " 100: ";
21         }
22         else {
23             cout << i * 10 << "-" << (i * 10)
24         }
25
26         ++i;
27
28         // print bar of asterisks
29         for (int stars{0}; stars < item; ++s
30             cout << '*';
31         }
32
33         cout << endl; // start a new line of
34     }
35 }
```

< >

Grade distribution:

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **

```
80-89: ****
90-99: **
100: *
```

Fig. 6.6 Printing a student grade distribution as a primitive bar chart.

Figure 6.6 reads the numbers from the `array` and graphs the information as a bar chart, displaying each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 16-24 output a grade range (e.g., "70-79: ") based on the current value of `i`. The nested `for` statement (lines 29-31) outputs the current bar as the appropriate number of asterisks. Note the loop-continuation condition in line 29 (`stars < item`). Each time the program reaches the inner `for`, the loop counts from 0 up to `item`, thus using a value in `array n` to determine the number of asterisks to display. In this example, elements 0-5 contain zeros because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges.

6.10 Using `array` Elements as Counters

11 Sometimes, programs use counter variables to summarize data, such as a survey's results. **Figure 5.3**'s die-rolling simulation used separate counters to track the number of occurrences of each side of a die as the program rolled the die 60,000,000 times. **Figure 6.7** reimplements that simulation using an array of counters. This version also uses the C++11 random-number generation capabilities that were introduced in [Section 5.10](#).

[Click here to view code image](#)

```
1 // fig06_07.cpp
2 // Die-rolling program using an array instead of separate counters
3 #include "fmt/format.h" // C++20: This will make printing easier
4 #include <iostream>
5 #include <array>
6 #include <random>
7 #include <ctime>
```

```

8  #include "gsl/gsl"
9  using namespace std;
10
11 int main() {
12     // use the default random-number genera-
13     // produce uniformly distributed pseudo-
14     default_random_engine engine(gsl::narrow-
15     uniform_int_distribution<int> randomInt
16
17     constexpr size_t arraySize{7}; // ignor-
18     array<int, arraySize> frequency{}; // i-
19
20     // roll die 60,000,000 times; use die v-
21     for (int roll{1}; roll <= 60'000'000; +
22         ++frequency.at(randomInt(engine)));
23    }
24
25    cout << fmt::format("{}{:>13}\n", "Face
26
27    // output each array element's value
28    for (size_t face{1}; face < frequency.s-
29        cout << fmt::format("{}{:>4}{:>13}\n",
30    }
31}

```



Face	Frequency
1	9997901
2	9999110
3	10001172
4	10003619
5	9997606
6	10000592

Fig. 6.7 Die-rolling program using an array instead of switch.

[Figure 6.7](#) uses the array `frequency` (line 18) to count the occurrences of die values. Line 22 of this program replaces the entire `switch` statement in lines 19-40 of [Fig. 5.3](#), by using a random value to determine which `frequency` element to increment for each die roll. The call `randomInt(engine)` produces a random subscript from 1 to 6, so `frequency` must be large enough to store six counters. We use a seven-element array in which we ignore element 0—it's clearer to have the die value 1 increment `frequency.at(1)` rather than `frequency.at(0)`. So, each face value is used directly as a `frequency` index. We also replace lines 44-49 of [Fig. 5.3](#) by looping through array `frequency` to output the results ([Fig. 6.7](#), lines 28-30).

6.11 Using arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:

Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being “awful” and 5 being “excellent.” Place the 20 responses in an integer array and determine the frequency of each rating.

This is a popular type of array-processing application ([Fig. 6.8](#)). We wish to summarize the number of responses of each rating (that is, 1-5). The array `responses` (lines 14-15) is a 20-element integer array of the students' responses to the survey. The array `responses` is declared `const`, as its values do not (and should not) change. We use a six- element array `frequency` (line 18) to count the number of occurrences of each response. Each element of the array is used as a counter for one of the survey responses and is initialized to zero. As in [Fig. 6.7](#), we ignore element 0.

[Click here to view code image](#)

```
1 // fig06_08.cpp
2 // Poll analysis program.
3 #include "fmt/format.h" // C++20: This wil
4 #include <iostream>
```

```

5 #include <array>
6 using namespace std;
7
8 int main() {
9     // define array sizes
10    constexpr size_t responseSize{20}; // si
11    constexpr size_t frequencySize{6}; // si
12
13    // place survey responses in array respo
14    const array<int, responseSize> responses
15        {1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3
16
17    // initialize frequency counters to 0
18    array<int, frequencySize> frequency{};

19
20    // for each answer, select responses ele
21    // as frequency index to determine elem
22    for (size_t answer{0}; answer < responseS
23        ++frequency.at(responses.at(answer));
24    }

25
26    cout << fmt::format("{}{:>12}\n", "Rating"
27
28    // output each array element's value
29    for (size_t rating{1}; rating < frequencyS
30        cout << fmt::format("{:>6}{:>12}\n",
31    }
32}

```

< >

Rating	Frequency
1	3
2	5
3	7
4	2
5	3

Fig. 6.8 Poll analysis program.

The first `for` statement (lines 22-24) takes one response at a time from the array `responses` and increments one of the counters `frequency.at(1)` to `frequency.at(5)`. The key statement in the loop is line 23, which increments the appropriate counter, depending on the value of `responses.at(answer)`.

Let's consider several iterations of the `for` loop. When control variable `answer` is 0, the value of `responses.at(answer)` is the value of `responses.at(0)` (i.e., 1 in line 15), so the program interprets `++frequency.at(responses(answer))` as

```
++frequency.at(1)
```

which increments the value in array element 1. To evaluate the expression in line 23, start with the value in the innermost set of square brackets (`answer`). Once you know `answer`'s value (which is the value of the control variable in line 22), plug it into the expression and evaluate the expression in the next outer set of square brackets—that is, `responses.at(answer)`, which is a value selected from the `responses` array in lines 14–15. Then use the resulting value as the index for the `frequency` array to specify which counter to increment.

When `answer` is 1, `responses.at(answer)` is the value of `responses.at(1)`, which is 2, so the program interprets `++frequency.at(responses(answer))` as

```
++frequency.at(2)
```

which increments array element 2.

When `answer` is 2, `responses[answer]` is the value of `responses.at(2)`, which is 5, so the program interprets `++frequency.at(responses(answer))` as

```
++frequency.at(5)
```

which increments array element 5, and so on. Regardless of the number of responses processed in the survey, the program requires only a six-element array (ignoring element zero) to summarize the results, because all the

response values are between 1 and 5 and the index values for a six-element array are 0 through 5.

6.12 Sorting and Searching arrays

In this section, we use the built-in C++ standard library **sort** function⁶ to arrange the elements in an array into ascending order and the built-in **binary_search** function to determine whether a value is in the array.

6. The C++ standard indicates that sort uses an $O(n \log n)$ algorithm, but does not specify which one.

Sorting data—placing it into ascending or descending order—is one of the most important computing applications. Virtually every organization must sort some data and, in many cases, massive amounts of it. Sorting is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science.

Often it may be necessary to determine whether an array contains a value that matches a certain **key value**. The process of finding a particular element of an array is called **searching**.

Demonstrating Functions **sort** and **binary_search**

Figure 6.9 begins by creating an unsorted array of strings (lines 11-12) and displaying the contents of the array (lines 16-18). Next, line 20 uses the C++ standard library function **sort** to sort the elements of the array **colors** into ascending order. The **sort** function’s arguments specify the range of elements that should be sorted—in this case, the entire array. The arguments **begin(colors)**, and **end(colors)** return “iterators” that represent the array’s beginning and end, respectively. Chapter 17 discusses iterators in depth. Functions **begin** and **end** are defined in **<array>**. As you’ll see, function **sort** can be used to sort the elements of several different types of data structures. Lines 24-26 display the contents of the sorted array.

[Click here to view code image](#)

```
1 // fig06_09.cpp
2 // Sorting and searching arrays.
```

```
3 #include <iostream>
4 #include <array>
5 #include <string>
6 #include <algorithm> // contains sort and :
7 using namespace std;
8
9 int main() {
10     constexpr size_t arraySize{7}; // size of
11     array<string, arraySize> colors{"red",
12         "green", "blue", "indigo", "violet"}
13
14     // output original array
15     cout << "Unsorted colors array:\n";
16     for (string color : colors) {
17         cout << color << " ";
18     }
19
20     sort(begin(colors), end(colors)); // so
21
22     // output sorted array
23     cout << "\nSorted colors array:\n";
24     for (string item : colors) {
25         cout << item << " ";
26     }
27
28     // search for "indigo" in colors
29     bool found{binary_search(begin(colors),
30         cout << "\n\n\"indigo\" " << (found ? "
31             << " found in colors array" << endl;
32
33     // search for "cyan" in colors
34     found = binary_search(begin(colors), en
35         cout << "\"cyan\" " << (found ? "was"
36             << " found in colors array" << endl;
37 }
```



```
Unsorted colors array:  
red orange yellow green blue indigo violet  
Sorted colors array:  
blue green indigo orange red violet yellow  
  
"indigo" was found in colors array  
"cyan" was not found in colors array
```

Fig. 6.9 Sorting and searching arrays.

Lines 29 and 34 use `binary_search` to determine whether a value is in the array. The sequence of values first must be sorted in ascending order—`binary_search` does not verify this for you. Performing a binary search on an unsorted array is a logic error that could lead to incorrect results. The function's first two arguments represent the range of elements to search, and the third argument is the search key—the value to find in the array. The function returns a `bool` indicating whether the value was found. In [Chapter 18](#), we'll use the C++ Standard function `find` to obtain the location of a search key in an array.

6.13 Multidimensional arrays

You can use arrays with two dimensions (i.e., indices) to represent **tables of values** consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify two indices—by convention, the first identifies the row and the second identifies the column. arrays that require two indices to identify a particular element are called **two-dimensional arrays** or **2-D arrays**. arrays with two or more dimensions are known as **multidimensional arrays**. The following diagram illustrates a two- dimensional array, `a`:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

array name
 Row subscript
 Column subscript

The array contains three rows and four columns, so it's said to be a 3-by-4 array. In general, an array with m rows and n columns is called an **m -by- n array**.

We have identified every element in the diagram above with an element name of the form $a[\text{row}][\text{column}]$. Similarly, you can access each element with `a.at(i).at(j)`,

`a.at(i).at(j)`

The elements names in row 0 all have a first index of 0; the elements names in column 3 all have a second index of 3. Referencing a two-dimensional array element as `a[x, y]` or `a.at(x, y)` is an error. Actually, `a[x, y]` and `a.at(x, y)` are treated as `a[y]` and `a.at(y)`, respectively, because C++ evaluates the expression `x, y` (containing a comma operator) simply as `y` (the last of the comma-separated expressions).

Figure 6.10 demonstrates initializing two-dimensional arrays in declarations. Lines 12-13 each declare an array of arrays with two rows and three columns.

[Click here to view code image](#)

```

1 // fig06_10.cpp
2 // Initializing multidimensional arrays.
3 #include <iostream>
4 #include <array>
```

```
5  using namespace std;
6
7  constexpr size_t rows{2};
8  constexpr size_t columns{3};
9  void printArray(const array<array<int, col
10
11 int main() {
12     const array<array<int, columns>, rows> a
13     const array<array<int, columns>, rows> a
14
15     cout << "Values in array1 by row are:" <
16     printArray(array1);
17
18     cout << "\nValues in array2 by row are:" <
19     printArray(array2);
20 }
21
22 // output array with two rows and three co
23 void printArray(const array<array<int, col
24     // loop through array's rows
25     for (auto const& row : a) {
26         // loop through columns of current r
27         for (auto const& element : row) {
28             cout << element << ' ';
29         }
30
31         cout << endl; // start new line of o
32     }
33 }
```



```
Values in array1 by row are:
1 2 3
4 5 6
```

```
Values in array2 by row are:
1 2 3
```

```
4 5 0
```

Fig. 6.10 Initializing multidimensional arrays.

Declaring an array of arrays

In the line 12 and line 13 declarations, notice the type:

```
array<array<int, columns>, rows>
```

The outer `array` type indicates that it has `rows` (2) elements of type

```
array<int, columns>
```

So, each of the outer array's element is an `array` of `ints` containing `columns` (3) elements.

Initializing an array of arrays

Line 12's `array1` declaration has six initializers. The compiler initializes row 0's elements, followed by row 1's elements. So, 1, 2 and 3 initialize row 0's elements, and 4, 5 and 6 initialize row 1's elements. Line 13's `array2` declaration provides only five initializers. The initializers are assigned to row 0, then row 1. Any elements that do not have an explicit initializer are initialized to zero, so `array2[1][2]` is 0.

Displaying an array of arrays

The program calls function `printArray` to output each array's elements. Notice that the function prototype (line 9) and definition (lines 23-33) specify that the function receives a two-row and three-column array. The parameter receives the `array` by reference and is declared `const` because the function does not modify the `array`'s elements.

Nested Range-Based for Statements

11 To process the elements of a two-dimensional `array`, we use a nested loop in which the outer loop iterates through the rows, and the inner loop iterates through the columns of a given row. Function `printArray`'s nested loop is implemented with range-based `for` statements. Lines 25 and 27 introduce the C++11 `auto` keyword, which tells the compiler to infer

(determine) a variable's data type based on the variable's initializer value. The outer loop's range variable `row` is initialized with an element from the parameter `a`. Looking at the array's declaration, you can see that it contains elements of type

```
array<int, columns>
```

so the compiler infers that `row` refers to a three-element array of `int` values (again, `columns` is 3). The `const&` in `row`'s declaration indicates that the reference cannot be used to modify the rows and prevents each row from being copied into the range variable. The inner loop's range variable `element` is initialized with one element of the array represented by `row`, so the compiler infers that `element` refers to an `int` because each row contains three `int` values. In many IDEs, hovering the mouse cursor over a variable declared with `auto` displays the variable's inferred type. Line 28 displays the value from a given row and column.

Nested Counter-Controlled `for` Statements

We could have implemented the nested loop with counter-controlled iteration as follows:

[Click here to view code image](#)

```
for (size_t row{0}; row < a.size(); ++row) {
    for (size_t column{0}; column < a.at(row).size();
        cout << a.at(row).at(column) << ' ';
    }
    cout << endl;
}
```



Initializing an array of arrays with a Fully Braced Initializer List

You may wish to initialize a two-dimensional array using separate initializer sublists for each row. For example, line 12 could be written as:

[Click here to view code image](#)

```
const array<array<int, columns>, rows> array1{
    {{1, 2, 3}, // row 0
     {4, 5, 6}} // row 1
```

```
};
```

If an initializer sublist has fewer elements than the row, the row's remaining elements would be initialized to 0.

Common Two-Dimensional array Manipulations: Setting a Columns' Values

The following `for` statement sets all the elements in row 2 of array `a` (from the beginning of this section) to zero:

[Click here to view code image](#)

```
for (size_t column{0}; column < 4; ++column) {  
    a.at(2).at(column) = 0; // a[2][column]  
}
```

The `for` statement varies only the second index (i.e., the column index). The preceding `for` statement is equivalent to the following assignment statements:

[Click here to view code image](#)

```
a.at(2)(0) = 0; // a[2][0]  
a.at(2)(1) = 0; // a[2][1]  
a.at(2)(2) = 0; // a[2][2]  
a.at(2)(3) = 0; // a[2][3]
```

Common Two-Dimensional array Manipulations: Totaling All the Elements with Nested Counter-Controlled `for` Loops

The following nested counter-controlled `for` statement totals the elements in the array `a` (that we defined in [Section 6.13](#)'s introduction):

[Click here to view code image](#)

```
total = 0;  
for (size_t row{0}; row < a.size(); ++row) {  
    for (size_t column{0}; column < a.at(row).size();  
        total += a.at(row).at(column); // a[row][column]  
    }  
}
```



The `for` statement totals the array’s elements one row at a time. The outer loop begins by setting the `row` index to 0, so the elements of row 0 may be totaled by the inner loop. The outer loop then increments `row` to 1, so the elements of row 1 can be totaled. Then, the outer `for` statement increments `row` to 2, so the elements of row 2 can be totaled.

Common Two-Dimensional array Manipulations: Totaling All the Elements with Nested Range-Based `for` Loops

Nested range-based `for` statements are the preferred way to implement the preceding loop:

[Click here to view code image](#)

```
total = 0;
for (auto row : a) { // for each row
    for (auto column : row) { // for each column in row
        total += column;
    }
}
```

6.14 Intro to Functional-Style Programming

 **PERF** Like other popular languages, such as Python, Java and C#, C++ supports several programming paradigms—procedural, object-oriented, generic (template-oriented) and “functional-style.” C++’s “functional-style” features help you write more concise code with fewer errors, and that’s easier to read, debug and modify. Functional-style programs also can be easier to parallelize to get better performance on today’s multi-core processors.

6.14.1 What vs. How

As a program’s tasks get more complicated, the code can become harder to read, debug and modify, and more likely to contain errors. Specifying *how* the code works can become complex. With functional-style programming, you specify *what* you want to do, and library code typically handles “the how” for you. This can eliminate many errors.

Consider the following range-based `for` statement from Fig. 6.5, which

totals the elements of the array `integers`:

[Click here to view code image](#)

```
for (const int item : integers) {  
    total += item;  
}
```

Though this procedural code hides the iteration details, we still have to specify how to total the elements by adding each `item` into the variable `total`. Each time you modify a variable, you can introduce errors. Functional-style programming emphasizes **immutability**— it avoids operations that modify variables' values. If this is your first exposure to functional-style programming, you might be wondering, “How can this be?” Read on.

Functional-Style Reduction with `accumulate`

Figure 6.11 replaces Fig. 6.5's range-based `for` statement with a call to the C++ standard library `accumulate` algorithm (from header `<numeric>`) . By default, this function knows *how* to reduce a container's values to the sum of those values. As we've mentioned, reduction is a key functional-style programming concept. The next example shows that you can customize how `accumulate` performs its reduction.

[Click here to view code image](#)

```
1 // fig06_11.cpp  
2 // Compute the sum of the elements of an a  
3 #include <array>  
4 #include <iostream>  
5 #include <numeric>  
6 using namespace std;  
7  
8 int main() {  
9     constexpr size_t arraySize{4};  
10    array<int, arraySize> integers{10, 20,  
11        cout << "Total of array elements: " <<  
12            accumulate(begin(integers), end(inte
```

```
13 }
```

The screenshot shows a code editor with a red box highlighting the line "Total of array elements: 100". Above the code, there is a line number "13" and a closing brace "}" on the right. Below the code, there is a grey scroll bar. The output window below the scroll bar contains the text "Total of array elements: 100" in a monospaced font.

Fig. 6.11 Compute the sum of the elements of an array.

Like function `sort` in [Section 6.12](#), function `accumulate`'s first two arguments specify the range of elements to total—in this case, all the elements from the beginning to the end of `integers`. The function *internally* increments the running total of the elements it processes, hiding those calculations. The third argument is that total's initial value.

Function `accumulate` uses **internal iteration**, which also is hidden from you. The function knows *how* to iterate through a range of elements and add each element to the running total. Stating *what* you want to do rather than programming *how* to do it is known as **declarative programming**—a key aspect of functional programming.

6.14.2 Passing Functions as Arguments to Other Functions—Introducing Lambda Expressions

Many standard library functions allow you to customize how they work by passing other functions as arguments. Functions that receive other functions as arguments are called **higher-order functions**—a key aspect of functional programming. For example, function `accumulate` totals elements by default. It also has an overloaded version, which receives as its fourth argument a function that specifies how to perform the reduction. Rather than simply totaling the values, [Fig. 6.12](#) calculates the product of the values.

[Click here to view code image](#)

```
1 // fig06_12.cpp
2 // Compute the product of an array's elements
3 #include <array>
4 #include <iostream>
5 #include <numeric>
```

```

6  using namespace std;
7
8  int multiply(int x, int y) {
9      return x * y;
10 }
11
12 int main() {
13     constexpr size_t arraySize{5};
14     array<int, arraySize> integers{1, 2, 3,
15
16     cout << "Product of integers: "
17         << accumulate(begin(integers), end(in
18
19     cout << "Product of integers with a lambd
20         << accumulate(begin(integers), end(in
21             [](const auto& x, const auto& y
22     )

```

◀ ▶

Product of integers: 120
 Product of integers with a lambda: 120

Fig. 6.12 Lambda expressions.

Calling `accumulate` with a Named Function

Line 17 calls `accumulate` for the array `integers` containing the values 1–5. We’re calculating the product of the values, so the third argument (i.e., the initial value of the reduction) is 1, rather than 0; otherwise, the final product would be 0. The fourth argument is the function to call for every array element—in this case, `multiply` (defined in lines 8–10). To calculate a product, this function must receive two arguments—the product so far and a value from the array—and must return a value, which becomes the new product. As `accumulate` iterates through the range of elements, it passes the current product and the next element as arguments. For this example, `accumulate` internally calls `multiply` five times:

- The first call passes the initial product (1) and the array’s first element (1), producing the product 1.
- The second call passes the current product (1) and the array’s second element (2), producing the product 2.
- The third call passes 2 and the array’s third element (3), producing the product 6.
- The fourth call passes 6 and the array’s fourth element (4), producing the product 24.
- The last call passes 24 and the array’s fifth element (5), producing the final result 120, which `accumulate` returns to its caller.

Calling `accumulate` with a Lambda Expression

11 In some cases, you may not need to reuse a function. In such cases, you can define a function where it’s needed by using a C++11 **lambda expression** (or simply **lambda**). A lambda expression is an *anonymous function*—that is, a function without a name. The call to `accumulate` in lines 20-21 uses the following lambda expression to perform the same task as `multiply` (line 17):

[Click here to view code image](#)

```
[] (const auto& x, const auto& y) {return x * y;}
```

Lambdas begin with the **lambda introducer** ([]), followed by a comma-separated parameter list and a function body. This lambda receives two parameters, calculates their product and returns the result.

14 You saw in [Section 6.13](#) that `auto` enables the compiler to infer a variable’s type based on its initial value. Specifying a lambda parameter’s type as `auto` enables the compiler to infer the parameter’s type, based on the context in which the lambda appears. In this example, `accumulate` calls the lambda once for each element of the `array`, passing the current product and the element’s value as the lambda’s arguments. Since the initial product (1) is an `int` and the `array` contains `ints`, the compiler infers the lambda parameters’ types as `int`. Using `auto` to infer each parameter’s type is a C++14 feature called **generic lambdas**. The compiler also infers the

lambda's return type from the expression `x * y`—both `x` and `y` are `ints`, so this lambda returns an `int`.

We declared the parameters as `const` references:

- They are `const` so the lambda's body does not modify the caller's variables.

PERF

- They are references for performance to ensure that the lambda does not copy large objects.

This lambda could be used with any numeric type that supports the multiplication operator. [Chapter 18](#) discusses lambda expressions in detail.

20

6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library

The C++ standard library has enabled functional-style programming for many years. C++20's new **ranges library**⁷ (header `<ranges>`) makes functional-style programming more convenient. In this section, we introduce two key aspects of this library—ranges and views:

7. At the time of this writing, only GNU C++ 10.1 implements the ranges library. We compiled and executed this example using the GNU GCC Docker container version 10.1 from https://hub.docker.com/_/gcc. Refer to the Chapter 1 test-drives for details on compiling and executing programs with this Docker container.
- A **range** is a collection of elements that you can iterate over, so an `array`, for example, is a range.
- A **view** enables you to specify an operation that manipulates a range. Views are **composable**—you can chain them together to process a range's elements through multiple operations.

The program of [Fig. 6.13](#) demonstrates several functional-style operations using C++20 ranges. We'll cover more features of this library in [Chapter 18](#).

[Click here to view code image](#)

```
1 // fig06_13.cpp
2 // Functional-style programming with C++20
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 #include <ranges>
7 using namespace std;
8
9 int main() {
10     // lambda to display results of range operations
11     auto showValues = [] (auto& values, const string message) {
12         cout << message << ":" << endl;
13
14         for (auto value : values) {
15             cout << value << " ";
16         }
17
18         cout << endl;
19     };
20 }
```



Fig. 6.13 Functional-style programming with C++20 ranges and views.

showValues Lambda for Displaying This Application's Results

17 Throughout this example, we display various range operations' results using a range-based `for` statement. Rather than repeating that code, we could define a function that receives a range and displays its values. For flexibility, we could use a function template, so it will work with ranges of various types. Here, we chose to define a generic lambda and demonstrate that you can store a lambda in a local variable (`showValues`; lines 11-19). You can 17 use the variable's name to call the lambda, as we do in lines 22, 27, 32, 38 and 49. C++17 made minor changes to the range-based `for` statement so that it eventually could be used with C++20 ranges, as we do in lines 14-16.

Generating a Sequential Range of Integers with `views::iota`

 **PERF** In many of this chapter's examples, we created an `array`, then processed its values. This required pre-allocating the array with the appropriate number of elements. In some cases, you can generate values *on demand*, rather than creating and storing the values in advance. Operations that generate values on demand use **lazy evaluation**, which can reduce your program's memory consumption and improve performance when all the values are not needed at once. Line 21 uses the `<range>` library's `views::iota` to generate a range of integers from its first argument (1) up to, but not including, its second argument (11). The values are not generated until the program iterates over the results, which occurs when we call `showValues` (line 22) to display the generated values.

[Click here to view code image](#)

```
21     auto values1 = views::iota(1, 11); // generates a range of integers from 1 to 10
22     showValues(values1, "Generate integers 1-10")
23
```

Generate integers 1-10: 1 2 3 4 5 6 7 8 9 10

Filtering Items with `views::filter`

A common functional-style programming operation is **filtering** elements to select only those that match a condition. This often produces fewer elements than the range being filtered. One way to implement filtering would be a loop that iterates over the elements, checks whether each element matches a condition, then do something with that element. That requires external iteration, which can be error-prone (as we discussed in [Section 6.6](#)).

With ranges and views, we can use `views::filter` to focus on *what* we want to accomplish—get the even integers in the range 1-10. The expression

[Click here to view code image](#)

```
values1 | views::filter([](const auto& x) { return
```

in line 26 uses the **| operator** to connect multiple operations. The first operation (`values1` from line 21) generates 1–10, and the second filters those results. Together, these operations form a **pipeline**. Each pipeline begins with a range, which is the data source (the values 1–10 produced by `values1`), followed by an arbitrary number of operations, each separated from the next by `|`.

[Click here to view code image](#)

```
24     // filter each value in values1, keeping only
25     auto values2 =
26     values1 | views::filter([](const auto& x)
27     showValues(values2, "Filtering even integers"
28
```

```
< [ ] >
```

Filtering even integers: 2 4 6 8 10

The argument to `views::filter` must be a function that receives one value to process and returns a `bool` indicating whether to keep the value. In this case, we passed a lambda that returns `true` if its argument is divisible by 2.

After lines 25–26 execute, `values2` represents a pipeline that can generate the integers 1–10 and filter those values, keeping only the even integers. The pipeline concisely represents *what* we want to do, but not *how* to implement it—`views::iota` knows *how* to generate integers, and `views::filter` knows *how* to use its function argument to determine whether to keep each value received from earlier in the pipeline.

When `showValues` iterates over `values2`, `views::iota` produces a value, then `views::filter` calls its argument to determine whether to keep the value. If so, the range-based `for` statement receives that value from the pipeline and displays it. Otherwise, the processing steps repeat with the next value generated by `views::iota`.

Mapping Items with `views::transform`

Another common functional-style programming operation is **mapping**

elements to new values (possibly of different types). Mapping produces a result with the same number of elements as the original data being mapped. With C++20 ranges, `views::transform` performs mapping operations. The pipeline in lines 30-31 adds another operation to the pipeline from lines 25-26, mapping the filtered results from `values2` to their squares with the lambda expression passed to `views::transform` in line 31.

[Click here to view code image](#)

```
29     // map each value in values2 to its square
30     auto values3 =
31         values2 | views::transform([](const auto&
32             showValues(values3, "Mapping even integers to
33
```

```
< > Mapping even integers to squares: 4 16 36 64 100
```

The argument to `views::transform` is a function that receives a value to process and returns the mapped value (possibly of a different type). When `showValues` iterates over the results of the `values3` pipeline:

- `views::iota` produces a value.
- `views::filter` determines whether the value is even and, if so, passes it to the next pipeline step; otherwise, processing proceeds with the next value generated by `views::iota`.
- `views::transform` calculates the square of the even integer (as specified by line 31's lambda), then the range-based `for` loop in `showValues` displays the value and processing proceeds with the next value generated by `views::iota`.

Combining Filtering and Mapping Operations into a Pipeline

A pipeline may contain an arbitrary number of operations separated by `|` operators. The pipeline in lines 35-37 combines all of the preceding operations into a single pipeline, and line 38 displays the results.

[Click here to view code image](#)

```
34 // combine filter and transform to get square
35 auto values4 =
36     values1 | views::filter([](const auto& x)
37         | views::transform([](const auto&
38             showValues(values4, "Squares of even integers
39
```

```
< > Squares of even integers: 4 16 36 64 100
```

Reducing a Range Pipeline with `accumulate`

C++ standard library functions like `accumulate` also work with range pipelines. Line 42 performs a reduction that sums the squares of the even integers produced by the pipeline in lines 35-37.

[Click here to view code image](#)

```
40 // total the squares of the even integers
41 cout << "Sum the squares of the even integers
42     accumulate(begin(values4), end(values4), 0
43
```

```
< > Sum the squares of the even integers from 2-10: 220
```

Filtering and Mapping an Existing Container's Elements

Various C++ containers, including arrays and vectors (Section 6.15), can be used as the data source in a range pipeline. Line 45 declares an array containing 1–10, then uses it in a pipeline that calculates the squares of the even integers in the array.

[Click here to view code image](#)

```
44 // process a container's elements
```

```
45     array<int, 10> numbers{1, 2, 3, 4, 5, 6, 7,
46     auto values5 =
47         numbers | views::filter([](const auto& x)
48                         | views::transform([](const auto&
49                         showValues(values5, "Squares of even integer
50 } }
```

```
< > Squares of even integers in array numbers: 4 16 36
```

6.15 Objects Natural Case Study: C++ Standard Library Class Template `vector`

We now continue our objects natural presentation by taking objects of C++ standard library class template `vector`⁸ “out for a spin.” A `vector` is similar to an `array`, but also supports dynamic resizing. `vector` is defined in header `<vector>` (Fig. 6.14, line 5) and belongs to namespace `std`. At the end of this section, we’ll demonstrate `vector`’s bounds checking capabilities (which `array` also has). There, we’ll introduce C++’s exception-handling mechanism, which can be used to detect and handle an out-of-bounds `vector` index. At that point, we’ll discuss the `<stdexcept>` header (line 6).

8. Chapter 17 discusses more `vector` capabilities.

[Click here to view code image](#)

```
1 // fig06_14.cpp
2 // Demonstrating C++ standard library clas
3 #include <iostream>
4 #include "fmt/format.h" // C++20: This wil
5 #include <vector>
6 #include <stdexcept>
7 using namespace std;
8
```

```
9     void outputVector(const vector<int>& items
10    void inputVector(vector<int>& items); // i:
11
```



Fig. 6.14 Demonstrating C++ standard library class template `vector`.

Creating `vector` Objects

Lines 13-14 create two `vector` objects that store values of type `int`—`integers1` contains seven elements, and `integers2` contains 10 elements. By default, all the elements of each `vector` object are set to 0. Like arrays, `vectors` can be defined to store most data types, by replacing `int` in `vector<int>` with the appropriate type.

[Click here to view code image](#)

```
12 int main() {
13     vector<int> integers1(7); // 7-element vect
14     vector<int> integers2(10); // 10-element ve
15
```



Notice that we used parentheses rather than braces to pass the size argument to each `vector` object's constructor. When creating a `vector`, if the braces contain one value of the `vector`'s element type, the compiler treats the braces as a one-element initializer list, rather than the `vector`'s size. So the following declaration actually creates a one-element `vector<int>` containing the `int` value 7, not a 7-element `vector`:

```
vector<int> integers1{7};
```

`vector` Member Function `size`; Function `outputVector`

Line 17 uses `vector` member function `size` to obtain `integers1`'s size (i.e., the number of elements). Line 19 passes `integers1` to function `outputVector` (lines 95-101), which displays the `vector`'s elements using a range-based `for`. Lines 22 and 24 perform the same tasks for `integers2`.

[Click here to view code image](#)

```
16 // print integers1 size and contents
17 cout << "Size of vector integers1 is " << int
18     << "\nvector after initialization:";
19 outputVector(integers1);
20
21 // print integers2 size and contents
22 cout << "\nSize of vector integers2 is " << i
23     << "\nvector after initialization:";
24 outputVector(integers2);
25
```

< >

```
Size of vector integers1 is 7
vector after initialization: 0 0 0 0 0 0 0
```

```
Size of vector integers2 is 10
vector after initialization: 0 0 0 0 0 0 0 0 0 0
```

Function `inputVector`

Lines 28-29 pass `integers1` and `integers2` to function `inputVector` (lines 104-108) to read values for each vector's elements from the user.

[Click here to view code image](#)

```
26 // input and print integers1 and integers2
27 cout << "\nEnter 17 integers:" << endl;
28 inputVector(integers1);
29 inputVector(integers2);
30
31 cout << "\nAfter input, the vectors contain:\"
32     << "integers1:";
33 outputVector(integers1);
34 cout << "integers2:";
```

```
35     outputVector(integers2);  
36
```

```
< ━━━━━━ >  
Enter 17 integers:  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

After input, the vectors contain:

```
integers1: 1 2 3 4 5 6 7  
integers2: 8 9 10 11 12 13 14 15 16 17
```

Comparing `vector` Objects for Inequality

In [Chapter 5](#), we introduced function overloading. A similar concept is operator overloading, which allows you to define how a built-in operator works for a custom type. The C++ standard library already defines overloaded operators `==` and `!=` for arrays and vectors, so you can compare two arrays or two vectors for equality or inequality, respectively. Line 40 compares two `vector` objects using the `!=` operator. This operator returns `true` if the contents of two `vectors` are not equal—that is, they have different lengths or the elements at the same index in both `vectors` are not equal. Otherwise, `!=` returns `false`.

[Click here to view code image](#)

```
37 // use inequality (!=) operator with vector o  
38 cout << "\nEvaluating: integers1 != integers2  
39  
40 if (integers1 != integers2) {  
41     cout << "integers1 and integers2 are not e  
42 }  
43
```

```
< ━━━━━━ >  
Evaluating: integers1 != integers2  
integers1 and integers2 are not equal
```

Initializing One `vector` with the Contents of Another

A new `vector` can be initialized with the contents of an existing `vector`. Line 46 creates a `vector` object `integers3` and initializes it with a copy of `integers1`. This invokes class template `vector`'s copy constructor to perform the copy operation. You'll learn about copy constructors in detail in [Chapter 14](#). Lines 48-50 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

[Click here to view code image](#)

```
44 // create vector integers3 using integers1 as
45 // initializer; print size and contents
46 vector<int> integers3{integers1}; // copy con
47
48 cout << "\nSize of vector integers3 is " << i
49     << "\nvector after initialization: ";
50 outputVector(integers3);
51
```

```
< ----- >
Size of vector integers3 is 7
vector after initialization: 1 2 3 4 5 6 7
```

Assigning `vectors` and Comparing `vectors` for Equality

Line 54 assigns `integers2` to `integers1`, demonstrating that the assignment (`=`) operator is overloaded to work with `vector` objects. Lines 56-59 output the contents of both objects to show that they now contain identical values. Line 64 then compares `integers1` to `integers2` with the equality (`==`) operator to determine whether the contents of the two objects are equal after the assignment (which they are).

[Click here to view code image](#)

```
52 // use overloaded assignment (=) operator
53 cout << "\nAssigning integers2 to integers1:"
54 integers1 = integers2; // assign integers2 to
```

```
55
56     cout << "integers1: ";
57     outputVector(integers1);
58     cout << "integers2: ";
59     outputVector(integers2);
60
61     // use equality (==) operator with vector obj
62     cout << "\nEvaluating: integers1 == integers2
63
64     if (integers1 == integers2) {
65         cout << "integers1 and integers2 are equal
66     }
67
```



```
Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal
```

Using the `at` Member Function to Access and Modify `vector` Elements

Lines 69 and 73 use `vector` member function `at` to obtain an element and use it as an *rvalue* and as an *lvalue*, respectively. Recall from [Section 4.12](#) that an *rvalue* cannot be modified, but an *lvalue* can. If the index is valid, member function `at` returns either

- a reference to the element at that location—this is a modifiable *lvalue* that can be used to change the value of the corresponding `vector` element, or
- a `const` reference to the element at that location—this is a nonmodifiable *lvalue* that cannot be used to change the value of the corresponding `vector` element.

A nonmodifiable *lvalue* is treated as a `const` object. If `at` is called on a `const` array or via a reference that's declared `const`, the function

returns a const reference.

[Click here to view code image](#)

```
68     // use square brackets to use the value at 1
69     cout << "\nintegers1.at(5) is " << integers1
70
71     // use square brackets to create lvalue
72     cout << "\n\nAssigning 1000 to integers1.at(
73     integers1.at(5) = 1000;
74     cout << "integers1: ";
75     outputVector(integers1);
76
```

```
integers1.at(5) is 13
```

```
Assigning 1000 to integers1.at(5)
integers1: 8 9 10 11 12 1000 14 15 16 17
```

As is the case with arrays, vectors also have a [] operator. C++ is not required to perform bounds checking when vector elements are accessed with square brackets. Therefore, you must ensure that operations using [] do not accidentally attempt to manipulate elements outside the bounds of the vector.

Exception Handling: Processing an Out-of-Range Index

Line 80 attempts to output the value in `integers1.at(15)`. The index 15 is outside the vector's bounds. Member function `at`'s bounds checking recognizes the invalid index and **throws** an **exception**, which indicates a runtime problem. The name “exception” suggests that the problem occurs infrequently. **Exception handling** enables you to create **fault-tolerant programs** that can process (or handle) exceptions. In many cases, this allows a program to continue executing as if no problems were encountered. For example, [Fig. 6.14](#) still runs to completion, even though we attempt to access an out-of-range index. More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user

of the problem, then terminate. Here we introduce exception handling briefly. You'll throw exceptions from your own custom functions in [Chapter 11](#). We'll discuss exception handling in detail in [Chapter 15](#).

[Click here to view code image](#)

```
77     // attempt to use out-of-range index
78     try {
79         cout << "\nAttempt to display integers1.a"
80         cout << integers1.at(15) << endl; // ERROR
81     }
82     catch (const out_of_range& ex) {
83         cerr << "An exception occurred: " << ex.what()
84     }
85
```

< >

```
Attempt to display integers1.at(15)
An exception occurred: invalid vector<T> subscript
```

< >

The **try** Statement

By default, an exception causes a C++ program to terminate. To handle an exception and possibly enable the program to continue executing, place any code that might throw an exception in a **try statement** (lines 78-84). The **try block** (lines 78-81) contains the code that might throw an exception, and the **catch block** (lines 82-84) contains the code that handles the exception if one occurs. As you'll see in [Chapter 15](#), you can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block. If the code in the try block executes successfully, lines 82-84 are ignored. The braces that delimit try and catch blocks' bodies are required.

Executing the **catch** Block

When the program calls `vector` member function `at` with the argument 15 (line 80), the function attempts to access the element at location 15, which is

outside the vector's bounds—`integers1` has only 10 elements at this point. Because bounds checking is performed at execution time, vector member function `at` generates an exception—specifically line 80 throws an `out_of_range` exception (from header `<stdexcept>`) to notify the program of this problem. At this point, the `try` block terminates immediately, and the `catch` block begins executing—if you declared any variables in the `try` block, they're now out of scope and are not accessible in the `catch` block.

 **PERF** The `catch` block declares a type (`out_of_range`) and an exception parameter (`ex`) that it receives as a reference. The `catch` block can handle exceptions of the specified type. Inside the block, you can use the parameter's identifier to interact with a caught exception object. Catching an exception by reference increases performance by preventing the exception object from being copied when it's caught.

what Member Function of the Exception Parameter

When lines 82-84 catch the exception, the program displays a message indicating the problem that occurred. Line 83 calls the exception object's `what` member function to get the exception object's error message and display it. Once the message is displayed in this example, the exception is considered handled, and the program continues with the next statement after the `catch` block's closing brace. In this example, lines 87-91 execute next. We use exception handling again in [Chapters 11-13](#), and [Chapter 15](#) presents a deeper look.

Changing the Size of a vector

One key difference between a `vector` and an `array` is that a `vector` can dynamically grow and shrink as the number of elements it needs to accommodate varies. To demonstrate this, line 87 shows the current size of `integers3`, line 88 calls the `vector`'s `push_back` member function to add a new element containing 1000 to the end of the `vector`, and line 89 shows the new size of `integers3`. Line 91 then displays `integers3`'s new contents.

[Click here to view code image](#)

```
86     // changing the size of a vector
87     cout << "\nCurrent integers3 size is: " << i
88     integers3.push_back(1000); // add 1000 to th
89     cout << "New integers3 size is: " << integer
90     cout << "integers3 now contains: ";
91     outputVector(integers3);
92 }
93
```

```
< >
Current integers3 size is: 7
New integers3 size is: 8
integers3 now contains: 1 2 3 4 5 6 7 1000
```

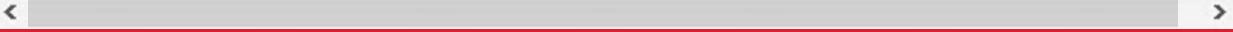
Functions `outputVector` and `inputVector`

Function `outputVector` uses a range-based `for` statement to obtain the value in each element of the vector for output. As with class template `array`, you could also do this using a counter-controlled loop and the `[]` operator, but the range-based `for` is recommended. Function `inputVector` uses a range-based `for` statement with an `int&` range variable, so it can be used to store an input value in the corresponding vector element.

[Click here to view code image](#)

```
94     // output vector contents
95     void outputVector(const vector<int>& items) {
96         for (const int item : items) {
97             cout << item << " ";
98         }
99         cout << endl;
100    }
101
102
103    // input vector contents
```

```
104 void inputVector(vector<int>& items) {  
105     for (int& item : items) {  
106         cin >> item;  
107     }  
108 }
```



C++11: List Initializing a `vector`

Many of the `array` examples in this chapter used list initializers to specify the initial `array` element values. C++11 also allows this for `vectors` (and other C++ standard library data structures).

Straight-Line Code

As you worked through this chapter's examples, you saw lots of `for` loops. But one thing you may have noticed in the first objects natural sections is that much of the code for creating and using objects is straight-line sequential code. Working with objects often flattens the coding process into lots of sequential member-function calls.

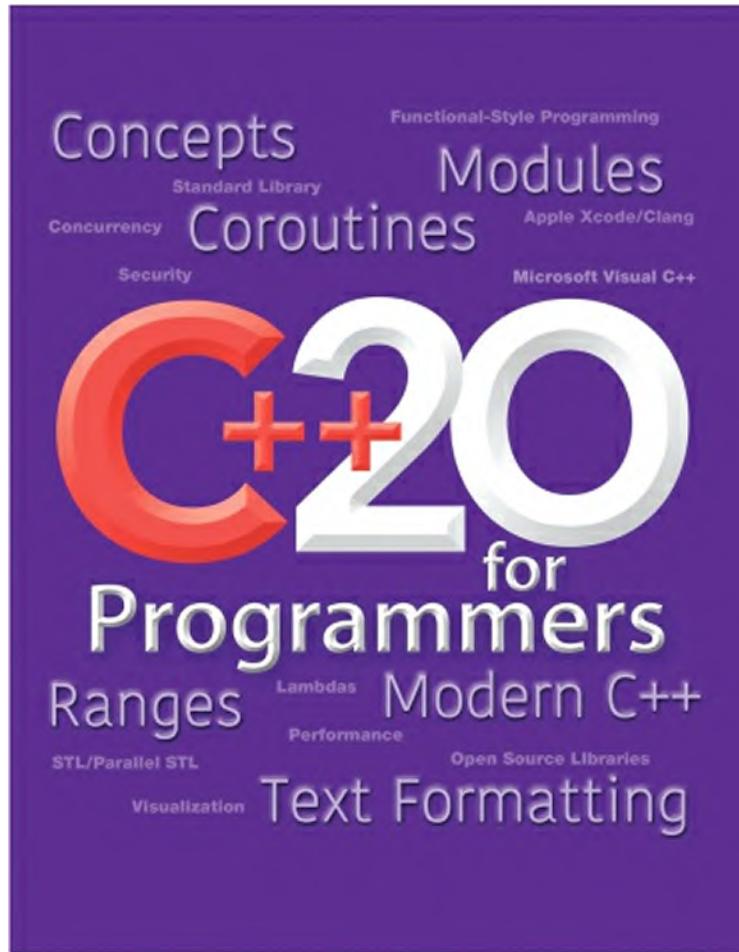
6.16 Wrap-Up

This chapter began our introduction to data structures, exploring the use of C++ standard library class templates `array` and `vector` to store data in and retrieve data from lists and tables of values. The chapter examples demonstrated how to declare an `array`, initialize an `array` and refer to individual elements of an `array`. We passed arrays to functions by reference and used the `const` qualifier to prevent the called function from modifying the `array`'s elements, thus enforcing the principle of least privilege. You learned how to use C++11's range-based `for` statement to manipulate all the elements of an `array`. We also showed how to use C++ standard library functions `sort` and `binary_search` to sort and search an `array`, respectively. You learned how to declare and manipulate multidimensional arrays of arrays. We used nested counter-controlled and nested range-based `for` statements to iterate through all the rows and columns of a two-dimensional `array`. We also showed how to use `auto` to infer a variable's type based on its initializer value. We introduced

functional-style programming in C++ using C++20’s new ranges. In later chapters, we’ll continue our coverage of data structures. In our first objects natural case study, we demonstrated the capabilities of the C++ standard library class template `vector`. In that example, we discussed how to access `array` and `vector` elements with bounds checking and demonstrated basic exception-handling concepts.

We’ve now introduced the basic concepts of classes, objects, control statements, functions, `array` objects and `vector` objects. In [Chapter 7](#), we present one of C++’s most powerful features—the pointer. Pointers keep track of where data and functions are stored in memory, which allows us to manipulate those items in interesting ways. As you’ll see, C++ also provides a language element called an array (different from the class template `array`) that’s closely related to pointers. In contemporary C++ code, its considered better practice to use C++11’s `array` class template rather than traditional arrays.

Chapter 7. (Downplaying) Pointers in Modern C++



Objectives

In this chapter, you'll:

- Learn what pointers are, and declare and initialize them.
- Use the address (`&`) and indirection (`*`) pointer operators.
- Compare the capabilities of pointers and references.
- Use pointers to pass arguments to functions by reference.
- Use pointer-based arrays and strings mostly in legacy code.

- Use `const` with pointers and the data they point to.
 - Use operator `sizeof` to determine the number of bytes that store a value of a particular type.
 - Understand pointer expressions and pointer arithmetic that you'll see in legacy code.
 - Use C++11's `nullptr` to represent pointers to nothing.
 - Use C++11's `begin` and `end` library functions with pointer-based arrays.
 - Learn various C++ Core Guidelines for avoiding pointers and pointer-based arrays to create safer, more robust programs.
 - Use C++20's `to_array` function to convert built-in arrays and initializer lists to `std::arrays`.
 - Continue our objects-natural approach by using C++20's class template `span` to create objects that are views into built-in arrays, `std::arrays` and `std::vectors`.
-

Outline

7.1 Introduction

7.2 Pointer Variable Declarations and Initialization

7.2.1 Declaring Pointers

7.2.2 Initializing Pointers

7.2.3 Null Pointers Prior to C++11

7.3 Pointer Operators

7.3.1 Address (`&`) Operator

7.3.2 Indirection (`*`) Operator

7.3.3 Using the Address (`&`) and Indirection (`*`) Operators

7.4 Pass-by-Reference with Pointers

7.5 Built-In Arrays

7.5.1 Declaring and Accessing a Built-In Array

7.5.2 Initializing Built-In Arrays

7.5.3 Passing Built-In Arrays to Functions

- 7.5.4 Declaring Built-In Array Parameters
 - 7.5.5 C++11: Standard Library Functions `begin` and `end`
 - 7.5.6 Built-In Array Limitations
 - 7.6 C++20: Using `to_array` to convert a Built-in Array to a `std::array`
 - 7.7 Using `const` with Pointers and the Data They Point To
 - 7.7.1 Using a Nonconstant Pointer to Nonconstant Data
 - 7.7.2 Using a Nonconstant Pointer to Constant Data
 - 7.7.3 Using a Constant Pointer to Nonconstant Data
 - 7.7.4 Using a Constant Pointer to Constant Data
 - 7.8 `sizeof` Operator
 - 7.9 Pointer Expressions and Pointer Arithmetic
 - 7.9.1 Adding Integers to and Subtracting Integers from Pointers
 - 7.9.2 Subtracting One Pointer from Another
 - 7.9.3 Pointer Assignment
 - 7.9.4 Cannot Dereference a `void*`
 - 7.9.5 Comparing Pointers
 - 7.10 Objects Natural Case Study: C++20 `spans`—Views of Contiguous Container Elements
 - 7.11 A Brief Intro to Pointer-Based Strings
 - 7.11.1 Command-Line Arguments
 - 7.11.2 Revisiting C++20's `to_array` Function
 - 7.12 Looking Ahead to Other Pointer Topics
 - 7.13 Wrap-Up
-

7.1 Introduction

This chapter discusses pointers, built-in pointer-based arrays and pointer-based strings (also called C-strings), each of which C++ inherited from the C programming language.

Downplaying Pointers in Modern C++

Pointers are powerful but challenging to work with and error-prone. So, Modern C++ (C++20, C++17, C++14 and C++11) has added features that

eliminate the need for most pointers. New software-development projects generally should prefer:

- using references to using pointers,
 - using `std::array`¹ and `std::vector` objects ([Chapter 6](#)) to using built-in pointer-based arrays, and
1. We pronounce “`std::`” as “standard,” so throughout this chapter we say “a `std::array`” rather than “an `std::array`,” which assumes “`std::`” is pronounced as its individual letters s, t and d.
- using `std::string` objects ([Chapters 2 and 8](#)) to pointer-based C-strings.

Sometimes Pointers Are Still Required

You’ll encounter pointers, pointer-based arrays and pointer-based strings frequently in the massive installed base of legacy C++ code. Pointers are required to:

- create and manipulate dynamic data structures, like linked lists, queues, stacks and trees that can grow and shrink at execution time—though most programmers will use the C++ standard library’s existing dynamic containers like `vector` and the containers we discuss in [Chapter 17](#),
- process command-line arguments, which a program receives as a pointer-based array of pointer-based C-strings, and
- pass arguments by reference if there’s a possibility of a `nullptr`² (i.e., a pointer to nothing; [Section 7.2.2](#))—a reference must refer to an actual object.

2. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-ptr-ref>.

Pointer-Related C++ Core Guidelines

CG We mention C++ Core Guidelines that encourage you to make your code safer and more robust by recommending you use techniques that avoid pointers, pointer-based arrays and pointer-based strings. For example, several guidelines recommend implementing pass-by-reference using references, rather than pointers.³

3. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-functions>.

C++20 Features for Avoiding Pointers

20 For programs that still require pointer-based arrays (e.g., command-line arguments), C++20 adds two new features that help make your programs safer and more robust:

- Function `to_array` converts a pointer-based array to a `std::array`, so you can take advantage of the features we demonstrated in [Chapter 6](#).
- `spans` offer a safer way to pass built-in arrays to functions. They're iterable, so you can use them with range-based `for` statements to conveniently process elements without risking out-of-bounds array accesses. Also, because `spans` are iterable, you can use them with standard library container-processing algorithms, such as `accumulate` and `sort`. We'll cover `spans` in this chapter's objects natural case study where you'll see that they also work with `std::array` and `std::vector`.

The key takeaway from reading this chapter is to avoid using pointers, pointer-based arrays and pointer-based strings whenever possible. If you must use them, take advantage of `to_array` and `spans`.

Other Concepts Presented in This Chapter

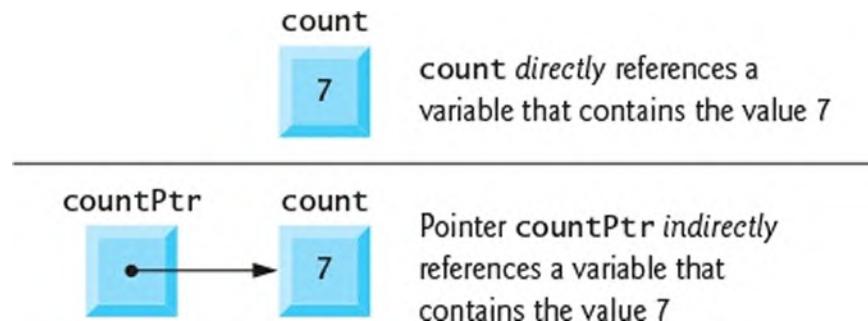
We declare and initialize pointers and demonstrate the pointer operators `&` and `*`. In [Chapter 5](#), we performed pass-by-reference with references. Here, we show that pointers also enable pass-by-reference. We demonstrate built-in, pointer-based arrays and their intimate relationship with pointers.

We show how to use `const` with pointers and the data they point to, and we introduce the `sizeof` operator to determine the number of bytes that store values of particular fundamental types and pointers. We demonstrate pointer expressions and pointer arithmetic.

C-strings were used widely in older C++ software. This chapter briefly introduces C-strings. You'll see how to process command-line arguments—a simple task for which C++ still requires you to use both pointer-based C-strings and pointer-based arrays.

7.2 Pointer Variable Declarations and Initialization

Pointer variables contain memory addresses as their values. Usually, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value** as shown in the following diagram:



Referencing a value through a pointer as in this diagram is called **indirection**.

7.2.1 Declaring Pointers

The following declaration declares the variable `countPtr` to be of type `int*` (i.e., a pointer to an `int` value) and is read (right-to-left), “`countPtr` is a pointer to an `int`”:

```
int* countPtr;
```

This `*` is not an operator; rather, it indicates that the variable to its right is a pointer. We like to include the letters `Ptr` in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.

7.2.2 Initializing Pointers

Security Initialize each pointer to `nullptr` (from C++11) or a memory address. A pointer with the value `nullptr` “points to nothing” and is known as a **null pointer**. From this point forward, when we refer to a “null pointer,” we mean a pointer with the value `nullptr`. Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.

7.2.3 Null Pointers Prior to C++11

In earlier C++ versions, the value specified for a null pointer was `0` or `NULL`.

NULL is defined in several standard library headers to represent the value 0. Initializing a pointer to NULL is equivalent to initializing it to 0, but prior to C++11, 0 was used by convention. The value 0 is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type (generally via a `reinterpret_cast`; Section 9.8).

7.3 Pointer Operators

The unary operators `&` and `*` create pointer values and “dereference” pointers, respectively. We show how to use these operators in the following sections.

7.3.1 Address (`&`) Operator

The **address operator (`&`)** is a unary operator that obtains the memory address of its operand. For example, assuming the declarations

[Click here to view code image](#)

```
int y{5}; // declare variable y
int* yPtr=nullptr; // declare pointer variable y
```

the following statement assigns the address of the variable `y` to pointer variable `yPtr`:

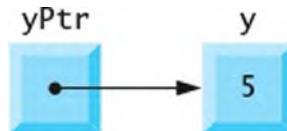
[Click here to view code image](#)

```
yPtr = &y; // assign address of y to yPtr
```

Variable `yPtr` is said to “point to” `y`—`yPtr` indirectly references the variable `y`’s value (5).

The `&` in the preceding statement is not a reference variable declaration, where `&` is always preceded by a type name. When declaring a reference, the `&` is part of the type. In an expression like `&y`, the `&` is the address operator.

The following diagram shows a memory representation after the previous assignment:



The “pointing relationship” is indicated by drawing an arrow from the box

that represents the pointer `yPtr` in memory to the box that represents the variable `y` in memory.

The following diagram shows another pointer memory representation with `int` variable `y` stored at memory location 600000 and pointer variable `yPtr` at location 500000



The address operator's operand must be an *lvalue*—the address operator cannot be applied to literals or to expressions that result in temporary values (like the results of calculations).

7.3.2 Indirection (*) Operator

Applying the unary *** operator** to a pointer results in an *lvalue* representing the object to which its pointer operand points. This operator is commonly referred to as the **indirection operator** or **dereferencing operator**. If `yPtr` points to `y` and `y` contains 5 (as in the preceding diagrams), the statement

```
cout << *yPtr << endl;
```

displays `y`'s value (5), as would the statement

```
cout << y << endl;
```

Using `*` in this manner is called **dereferencing a pointer**. A dereferenced pointer also can be used as an *lvalue* in an assignment. The following assigns 9 to `y`:

```
*yPtr = 9;
```

In this statement, `*yPtr` is an alias for `y`. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

which places the input value in `y`.

Undefined Behaviors

 **Security** Dereferencing an uninitialized pointer results in undefined

behavior that could cause a fatal execution-time error. This also could lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results. This is a potential security flaw that an attacker might be able to exploit to access data, overwrite data or even execute malicious code.^{4,5,6} Dereferencing a null pointer results in undefined behavior and typically causes a fatal execution-time error. In industrial-strength code, ensure that a pointer is not `nullptr` before dereferencing it.⁷

4. “Undefined Behavior.” Wikipedia. Wikimedia Foundation, May 30, 2020. https://en.wikipedia.org/wiki/Undefined_behavior.
5. “Common Weakness Enumeration.” CWE. Accessed June 14, 2020. <https://cwe.mitre.org/data/definitions/824.html>.
6. “Dangling Pointer.” Wikipedia. Wikimedia Foundation, June 8, 2020. https://en.wikipedia.org/wiki/Dangling_pointer.
7. The C++ Core Guidelines recommend using the `gsl::not_null` class template from the Guidelines Support Library (GSL) to declare pointers that should not have the value `nullptr`. Throughout this book, we adhere to the C++ Core Guidelines as appropriate. At the time of this writing, the Guidelines Support Library’s `gsl::not_null` implementation did not produce helpful error messages in our compilers, so we chose not to use `gsl::not_null` in our code.

7.3.3 Using the Address (`&`) and Indirection (`*`) Operators

Figure 7.1 demonstrates the `&` and `*` pointer operators, which have the third-highest level of precedence (see the Appendix A for the complete operator-precedence chart). Memory locations are output by `<<` in this example as hexadecimal (i.e., base-16) integers. (See Appendix D, Number Systems, for more information on hexadecimal integers.) The output shows that variable `a`’s address (line 10) and `aPtr`’s value (line 11) are identical, confirming that `a`’s address was indeed assigned to `aPtr` (line 8). The outputs from lines 12–13 confirm that `*aPtr` has the same value as `a`. The memory addresses output by this program with `cout` and `<<` are compiler- and platform-dependent, and typically change with each program execution, so you’ll likely see different addresses.

[Click here to view code image](#)

```
1 // fig07_01.cpp
2 // Pointer operators & and *.
```

```

3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     constexpr int a{7}; // initialize a with 7
8     const int* aPtr = &a; // initialize aPtr to point to a
9
10    cout << "The address of a is " << &a
11        << "\nThe value of aPtr is " << aPtr;
12    cout << "\n\nThe value of a is " << a
13        << "\n\nThe value of *aPtr is " << *aPtr;
14 }

```

The address of a is 002DFD80
The value of aPtr is 002DFD80

The value of a is 7
The value of *aPtr is 7

Fig. 7.1 Pointer operators `&` and `*`.

7.4 Pass-by-Reference with Pointers

There are three ways in C++ to pass arguments to a function:

- pass-by-value
- pass-by-reference with a reference argument
- **pass-by-reference with a pointer argument.**

 **PERF** Chapter 5 showed the first two. Here, we explain pass-by-reference with a pointer. Pointers, like references, can be used to modify variables in the caller or to pass large data objects by reference to avoid the overhead of copying objects. You accomplish pass-by-reference via pointers and the indirection operator (`*`). When calling a function that receives a pointer, pass a variable's address by applying the address operator (`&`) to the

variable's name.

An Example of Pass-By-Value

Figures 7.2 and 7.3 present two functions that each cube an integer. Figure 7.2 passes variable number by value (line 12) to function cubeByValue (lines 17–19), which cubes its argument and passes the result back to main using a return statement (line 18). We stored the new value in number (line 12), though that is not required. For instance, the calling function might want to examine the function call's result before modifying variable number.

[Click here to view code image](#)

```
1 // fig07_02.cpp
2 // Pass-by-value used to cube a variable's
3 #include <iostream>
4 using namespace std;
5
6 int cubeByValue(int n); // prototype
7
8 int main() {
9     int number{5};
10
11     cout << "The original value of number is "
12     number = cubeByValue(number); // pass n
13     cout << "\nThe new value of number is "
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
18     return n * n * n; // cube local variable
19 }
```



```
The original value of number is 5
The new value of number is 125
```

Fig. 7.2 Pass-by-value used to cube a variable's value.

[Click here to view code image](#)

```
1 // fig07_03.cpp
2 // Pass-by-reference with a pointer argument
3 // variable's value.
4 #include <iostream>
5 using namespace std;
6
7 void cubeByReference(int* nPtr); // prototype
8
9 int main() {
10     int number{5};
11
12     cout << "The original value of number is "
13     cubeByReference(&number); // pass number
14     cout << "\nThe new value of number is "
15 }
16
17 // calculate cube of *nPtr; modifies variable
18 void cubeByReference(int* nPtr) {
19     *nPtr = *nPtr * *nPtr * *nPtr; // cube
20 }
```



```
The original value of number is 5
The new value of number is 125
```

Fig. 7.3 Pass-by-reference with a pointer argument used to cube a variable's value.

An Example of Pass-By-Reference with Pointers

Figure 7.3 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument (line 13)—the address of

number is passed to the function. Function cubeByReference (lines 18–20) specifies parameter nPtr (a pointer to int) to receive its argument. The function uses the dereferenced pointer—`*nPtr`, an alias for number in main—to cube the value to which nPtr points (line 19). This directly changes the value of number in main (line 10). Line 19 can be made clearer with redundant parentheses:

[Click here to view code image](#)

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube *nPt:  
  <                                        >
```

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, function cubeByReference’s header (line 18) specifies that the function receives a pointer to an int as an argument, stores the address in nPtr and does not return a value.

Insight: Pass-By-Reference with a Pointer Actually Passes the Pointer By Value

Passing a variable by reference with a pointer does not actually pass anything by reference. Rather, a pointer to that variable is passed by value. That pointer value is copied into the function’s corresponding pointer parameter. The called function can then access the caller’s variable by dereferencing the pointer, thus accomplishing pass-by-reference.

Graphical Analysis of Pass-By-Value and Pass-By-Reference

Figures 7.4–7.5 graphically analyze the execution of Fig. 7.2 and Fig. 7.3, respectively. The rectangle above a given expression or variable contains the value being produced by a step in the diagram. Each diagram’s right column shows functions cubeByValue (Fig. 7.2) and cubeByReference (Fig. 7.3) only when they’re executing.

Step 1: Before main calls cubeByValue:

```
int main() {                                number  
    int number{5};                         5  
    number = cubeByValue(number);  
}
```

Step 2: After cubeByValue receives the call:

```
int main() {                                number  
    int number{5};                         5  
    number = cubeByValue(number);  
}                                              n  
int cubeByValue( int n ) {  
    return n * n * n;                      5  
}
```

Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:

```
int main() {                                number  
    int number{5};                         5  
    number = cubeByValue(number);  
}                                              n  
int cubeByValue(int n) {  
    return n * n * n;                      125  
}
```

Step 4: After cubeByValue returns to main and before assigning the result to number:

```
int main() {                                number  
    int number{5};                         5  
    number = cubeByValue(number);  
}
```

Step 5: After main completes the assignment to number:

```
int main() {                                number  
    int number{5};                         125  
    number = cubeByValue(number);  
}
```

Fig. 7.4 Pass-by-value analysis of the program of Fig. 7.2.

Step 1: Before main calls cubeByReference:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

5

Step 2: After cubeByReference receives the call and before *nPtr is cubed:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

5

```
void cubeByReference( int* nPtr ) {  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}  
  
call establishes this pointer
```

nPtr

Step 3: Before *nPtr is assigned the result of the calculation $5 * 5 * 5$:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

5

```
void cubeByReference(int* nPtr) {  
    125  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}  
  
nPtr
```

nPtr

Step 4: After *nPtr is assigned 125 and before program control returns to main:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

125

```
void cubeByReference(int* nPtr) {  
    125  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}  
  
called function modifies caller's  
variable
```

nPtr

Step 5: After cubeByReference returns to main:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

125

Fig. 7.5 Pass-by-reference analysis of the program of Fig. 7.3.

7.5 Built-In Arrays

 **Security** Here we present built-in arrays, which like `std::arrays` are also fixed-size data structures. We include this presentation mostly because you'll see built-in arrays in legacy C++ code. New applications should use `std::array` and `std::vector` to create safer, more robust applications.

20 20 In particular, `std::array` and `std::vector` objects always know their own size—even when passed to other functions, which is not the case for built-in arrays. If you work on applications containing built-in arrays, you can use C++20's `to_array` function to convert them to `std::arrays` (Section 7.6), or you can process them more safely using C++20's spans (Section 7.10). There are some cases in which built-in arrays are required, such as receiving command-line arguments, which we demonstrate in Section 7.11.

7.5.1 Declaring and Accessing a Built-In Array

As with `std::array`, you must specify a built-in array's element type and number of elements, but the syntax is different. For example, to reserve five elements for a built-in array of ints named `c`, use

[Click here to view code image](#)

```
int c[5]; // c is a built-in array of 5 integers
```

You use the subscript `([])` operator to access a built-in array's elements. Recall from Chapter 6 that the subscript `([])` operator does not provide bounds checking for `std::arrays`—this is also true for built-in arrays. Of course, you can use `std::array`'s `at` member function to do bounds checking.

7.5.2 Initializing Built-In Arrays

You can initialize the elements of a built-in array using an initializer list. For example,

```
int n[5]{50, 20, 30, 10, 40};
```

creates and initializes a built-in array of five `int`s. If you provide fewer initializers than the number of elements, the remaining elements are **value initialized**—fundamental numeric types are set to 0, `bool`s are set to `false`, pointers are set to `nullptr` and, as we'll see in [Chapter 10](#), objects receive the default initialization specified by their class definitions. If you provide too many initializers, a compilation error occurs.

The compiler can size a built-in array by counting an initializer list's elements. For example, the following creates a five-element array:

```
int n[] {50, 20, 30, 10, 40};
```

7.5.3 Passing Built-In Arrays to Functions

The value of a built-in array's name is implicitly convertible to a `const` or non-`const` pointer to the built-in array's first element—this is known as **decaying to a pointer**. So the array name `n` above is equivalent to `&n[0]`, which is a pointer to the element containing 50. You don't need to take the address (`&`) of a built-in array to pass it to a function—you simply pass its name. As you saw in [Section 7.4](#), a function that receives a pointer to a variable in the caller can modify that variable in the caller. For built-in arrays, the called function can modify all the elements in the caller—unless the parameter is declared `const`. Applying `const` to a built-in array parameter to prevent the argument array in the caller from being modified in the called function is another example of the principle of least privilege.

7.5.4 Declaring Built-In Array Parameters

You can declare a built-in array parameter in a function header, as follows:

[Click here to view code image](#)

```
int sumElements(const int values[], size_t number)
```



which indicates that the function's first argument should be a one-dimensional built-in array of `int`s that should not be modified by the function. Unlike `std::arrays` and `std::vectors`, built-in arrays don't know their own size, so a function that processes a built-in array should also

receive the built-in array's size.

The preceding header also can be written as

[Click here to view code image](#)

```
int sumElements(const int* values, size_t numberO:
```



The compiler does not differentiate between a function that receives a pointer and a function that receives a built-in array. In fact, the compiler converts `const int values[]` to `const int* values` under the hood. This means the function must “know” when it’s receiving a built-in array vs. a single variable that’s being passed by reference.

CG 20 The C++ Core Guidelines specifically say not to pass built-in arrays to functions;⁸ rather, you should pass C++20 spans because they maintain a pointer to the array’s first element and the array’s size. In [Section 7.10](#), we’ll demonstrate spans and you’ll see that passing a span is superior to passing a built-in array and its size to a function.

8. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array>.

7.5.5 C++11: Standard Library Functions `begin` and `end`

11 In [Section 6.12](#), we sorted a `std::array` of strings called `colors` as follows:

[Click here to view code image](#)

```
sort(begin(colors), end(colors)); // sort contents:
```



Functions `begin` and `end` specified that the entire `std::array` should be sorted. Function `sort` (and many other C++ Standard Library functions) also can be applied to built-in arrays. For example, to sort the built-in array `n` ([Section 7.5.2](#)), you can write

[Click here to view code image](#)

```
sort(begin(n), end(n)); // sort contents of built-
```



20 For a built-in array, `begin` and `end` work only in the scope that originally defines the array, which is where the compiler knows the array's size. Again, you should pass built-in arrays to other functions using C++20 spans, which we demonstrate in [Section 7.10](#).

7.5.6 Built-In Array Limitations

Built-in arrays have several limitations:

- They cannot be compared using the relational and equality operators—you must use a loop to compare two built-in arrays element by element. If you had two `int` arrays named `array1` and `array2`, the condition `array1 == array2` would always be false, even if the arrays' contents are identical. Remember, array names decay to `const` pointers to the arrays' first elements. And, of course, for separate arrays, those elements reside at different memory locations.
- They cannot be assigned to one another—an array name is effectively a `const` pointer, so it can't be changed by assignment.
- They don't know their own size—a function that processes a built-in array typically receives both the built-in array's name and its size as arguments.
- They don't provide automatic bounds checking—you must ensure that array-access expressions use subscripts within the built-in array's bounds.

7.6 C++20: Using `to_array` to Convert a Built-in Array to a `std::array`

20 CG  **Security** In industry, you'll encounter C++ legacy code that uses built-in arrays. The C++ Core Guidelines say you should prefer `std::arrays` and `std::vectors` to built-in arrays because they're safer, and they do not become pointers when you pass them to functions.⁹ C++20's new `to_array` function¹⁰ (header `<array>`) makes it convenient to create a `std::array` from a built-in array or an initializer list. [Figure 7.6](#) demonstrates `to_array`. We use a generic lambda expression (lines 9–13)

to display each `std::array`'s contents. Again, specifying a lambda parameter's type as `auto` enables the compiler to infer the parameter's type, based on the context in which the lambda appears. In this program, the generic lambda automatically determines the type of the `std::array` over which it iterates.

9. C++ Core Guidelines. Accessed June 14, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-arrays>.
10. “`to_array` From LFTS with Updates.” `to_array` from LFTS with updates—HackMD. Accessed June 14, 2020. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0325r4.html>.

[Click here to view code image](#)

```
1 // fig07_06.cpp
2 // C++20: Creating std::arrays with to_array
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     // lambda to display a collection of items
9     const auto display = [](const auto& item) {
10         for (const auto& item : items) {
11             cout << item << " ";
12         }
13     };
14 }
```



Fig. 7.6 C++20: Creating `std::arrays` with `to_array`.

Using `to_array` to create a `std::array` from a Built-In Array

Line 18 creates a three-element `std::array` of `ints` by copying the contents of built-in array `values1`. We use `auto` to infer the `std::array` variable's type and size. If we declare the array's type and

size explicitly and it does not match `to_array`'s return value, a compilation error occurs. We assign the result to the variable `array1`. Lines 20 and 21 display the `std::array`'s size and contents to confirm that it was created correctly.

[Click here to view code image](#)

```
15 const int values1[3]{10, 20, 30};  
16  
17 // creating a std::array from a built-in array  
18 const auto array1 = to_array(values1);  
19  
20 cout << "array1.size() = " << array1.size() <<  
21 display(array4); // use lambda to display cont  
22
```

< >

```
array1.size() = 3  
array1: 10 20 30
```

Using `to_array` to create a `std::array` from an Initializer List

Line 24 shows that `to_array` can create a `std::array` from an initializer list. Lines 25 and 26 display the array's size and contents to confirm that it was created correctly.

[Click here to view code image](#)

```
23 // creating a std::array from an initializer  
24 const auto array2 = to_array({1, 2, 3, 4});  
25 cout << "\n\narray2.size() = " << array2.size()  
26 display(array2); // use lambda to display co  
27  
28 cout << endl;  
29
```

< >

```
array2.size() = 4  
array2: 1 2 3 4
```

7.7 Using `const` with Pointers and the Data Pointed To

This section discusses how to combine `const` with pointer declarations to enforce the principle of least privilege. Chapter 5 explained that pass-by-value copies an argument's value into a function's parameter. If the copy is modified in the called function, the original value in the caller does not change. In some instances, even the copy of the argument's value should not be altered in the called function.

If a value does not (or should not) change in the body of a function to which it's passed, declare the parameter `const`. Before using a function, check its function prototype to determine the parameters that it can and cannot modify.

There are four ways to pass a pointer to a function:

- a nonconstant pointer to nonconstant data,
- a nonconstant pointer to constant data (Fig. 7.7),
- a constant pointer to nonconstant data (Fig. 7.8) and
- a constant pointer to constant data (Fig. 7.9).

Each combination provides a different level of access privilege.

7.7.1 Using a Nonconstant Pointer to Nonconstant Data

The highest privileges are granted by a **nonconstant pointer to nonconstant data**:

- the data can be modified through the dereferenced pointer, and
- the pointer can be modified to point to other data.

Such a pointer's declaration (e.g., `int* countPtr`) does not include `const`.

7.7.2 Using a Nonconstant Pointer to Constant Data

A **nonconstant pointer to constant data** is

- a pointer that can be modified to point to any data of the appropriate type, but
- the data to which it points cannot be modified through that pointer.

The declaration for such a pointer places `const` to the left of the pointer's type, as in¹¹

¹¹. Some programmers prefer to write this as `int const* countPtr;`. They'd read this declaration from right to left as "countPtr is a pointer to a constant integer."

```
const int* countPtr;
```

The declaration is read from right to left as "countPtr is a pointer to an integer constant" or, more precisely, "countPtr is a nonconstant pointer to an integer constant."

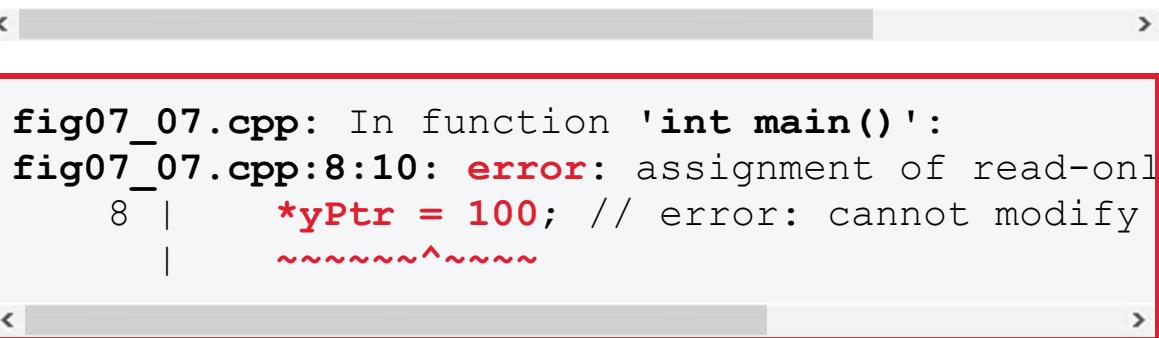
Figure 7.7 demonstrates the GNU C++ compilation error produced when you try to modify data via a nonconstant pointer to constant data.

[Click here to view code image](#)

```

1 // fig07_07.cpp
2 // Attempting to modify data through a
3 // nonconstant pointer to constant data.
4
5 int main() {
6     int y{0};
7     const int* yPtr{&y};
8     *yPtr = 100; // error: cannot modify a
9 }
```

GNU C++ compiler error message:



```

fig07_07.cpp: In function 'int main()':
fig07_07.cpp:8:10: error: assignment of read-only
  8 |     *yPtr = 100; // error: cannot modify
  | ~~~~~^~~~~~
```

Fig. 7.7 Attempting to modify data through a nonconstant pointer to `const` data.

  **PERF Security** Use pass-by-value to pass fundamental-type arguments (e.g., `ints`, `doubles`, etc.) unless the called function must directly modify the value in the caller. This is another example of the principle of least privilege. If large objects do not need to be modified by a called function, pass them using references to constant data or using pointers to constant data—though references are preferred. This gives the performance benefits of pass-by-reference and avoids the copy overhead of pass-by-value. Passing large objects using references to constant data or pointers to constant data also offers the security of pass-by-value.

7.7.3 Using a Constant Pointer to Nonconstant Data

A **constant pointer to nonconstant data** is a pointer that

- always points to the same memory location, and
- the data at that location can be modified through the pointer.

Pointers that are declared `const` must be initialized when they're declared, but if the pointer is a function parameter, it's initialized with the pointer that's passed to the function. Each successive call to the function reinitializes that function parameter.

Figure 7.8 attempts to modify a constant pointer. Line 9 declares pointer `ptr` to be of type `int* const`. The declaration is read from right to left as “`ptr` is a constant pointer to a nonconstant integer.” The pointer is initialized with the address of integer variable `x`. Line 12 attempts to assign the address of `y` to `ptr`, but the compiler generates an error message. No error occurs when line 11 assigns the value 7 to `*ptr`. The nonconstant value to which `ptr` points can be modified using the dereferenced `ptr`, even though `ptr` itself has been declared `const`.

[Click here to view code image](#)

```
1 // fig07_08.cpp
```

```

2 // Attempting to modify a constant pointer
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer
8     // through ptr, but ptr always points to x
9     int* const ptr{&x}; // const pointer must
10
11     *ptr = 7; // allowed: *ptr is not const
12     ptr = &y; // error: ptr is const; cannot
13 }
```

Microsoft Visual C++ compiler error message:

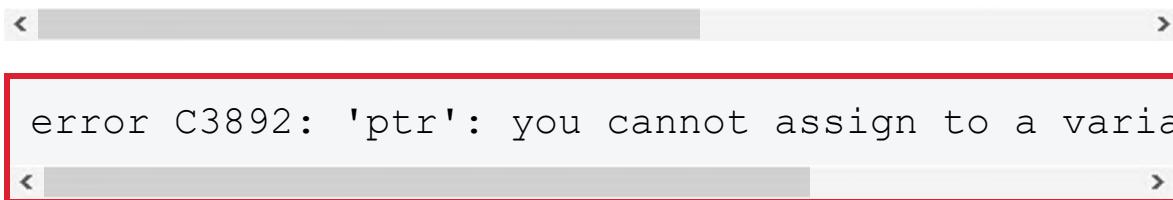


Fig. 7.8 Attempting to modify a constant pointer to nonconstant data.

7.7.4 Using a Constant Pointer to Constant Data

The minimum access privileges are granted by a **constant pointer to constant data**:

- such a pointer always points to the same memory location, and
- the data at that location cannot be modified via the pointer.

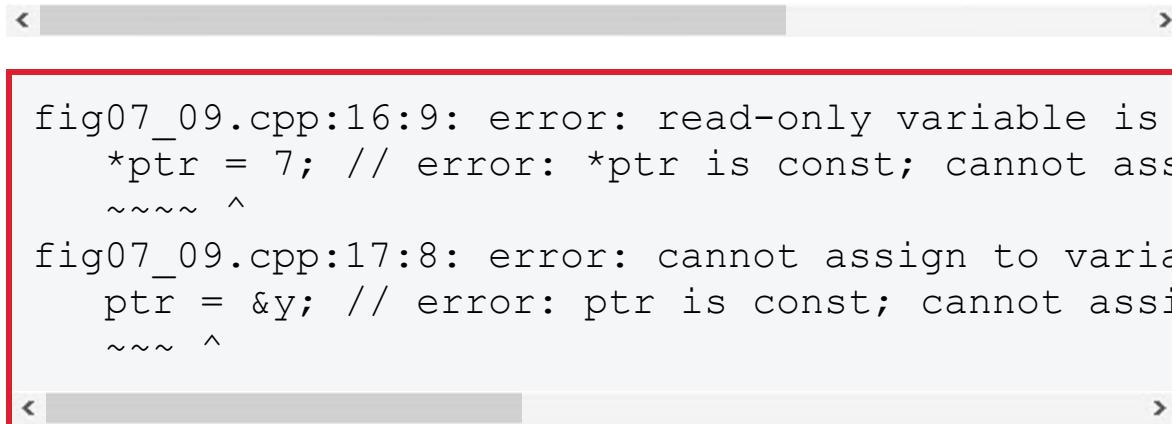
Figure 7.9 declares pointer variable `ptr` to be of type `const int* const` (line 12). This declaration is read from right to left as “`ptr` is a constant pointer to an integer constant.” The figure shows the Apple Clang compiler’s error messages for attempting to modify the data to which `ptr` points (line 16) and attempting to modify the address stored in the pointer variable (line 17). In line 14, no errors occur, because *neither* the pointer *nor* the data it points to is being modified.

[Click here to view code image](#)

```

1 // fig07_09.cpp
2 // Attempting to modify a constant pointer
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int x{5}, y;
8
9     // ptr is a constant pointer to a const.
10    // ptr always points to the same location.
11    // at that location cannot be modified.
12    const int* const ptr{&x};
13
14    cout << *ptr << endl;
15
16    *ptr = 7; // error: *ptr is const; cannot assign
17    ptr = &y; // error: ptr is const; cannot assign
18 }
```

Apple Clang compiler error messages:



```

fig07_09.cpp:16:9: error: read-only variable is
    *ptr = 7; // error: *ptr is const; cannot assign
    ~~~~ ^
fig07_09.cpp:17:8: error: cannot assign to variable
    ptr = &y; // error: ptr is const; cannot assign
    ~~~ ^
```

Fig. 7.9 Attempting to modify a constant pointer to constant data.

7.8 **sizeof** Operator

The compile-time unary operator **sizeof** determines the size in bytes of a built-in array or of any other data type, variable or constant *during program*

compilation. When applied to a built-in array's name, as in Fig. 7.10¹² (line 12), `sizeof` returns the total number of bytes in the built-in array as a value of type `size_t`. The computer we used to compile this program stores `double` variables in 8 bytes of memory. `numbers` is declared to have 20 elements (line 10), so it uses 160 bytes in memory. Applying `sizeof` to a pointer parameter (line 20) in a function that receives a built-in array, returns the size of the pointer in bytes (4 on the system we used), not the built-in array's size. Using the `sizeof` operator in a function to find the size in bytes of a built-in array parameter returns the size in bytes of a pointer, not the size in bytes of the built-in array.

12. This is a mechanical example to demonstrate how `sizeof` works. If you use static code-analysis tools, such as the C++ Core Guidelines checker in Microsoft Visual Studio, you'll receive warnings because you should not pass built-in arrays to functions.

[Click here to view code image](#)

```
1 // fig07_10.cpp
2 // Sizeof operator when used on a built-in
3 // returns the number of bytes in the buil
4 #include <iostream>
5 using namespace std;
6
7 size_t getSize(double* ptr); // prototype
8
9 int main() {
10    double numbers[20]; // 20 doubles; occup
11
12    cout << "The number of bytes in the arra
13
14    cout << "\nThe number of bytes returned :
15        << getSize(numbers) << endl;
16 }
17
18 // return size of ptr
19 size_t getSize(double* ptr) {
20    return sizeof(ptr);
```

```
21 }
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 7.10 `sizeof` operator when applied to a built-in array's name returns the number of bytes in the built-in array.

Determining the Sizes of the Fundamental Types, a Built-In Array and a Pointer

11 Figure 7.11 uses `sizeof` to calculate the number of bytes used to store various standard data types. The output was produced using the Apple Clang compiler in Xcode. Type sizes are platform dependent. When we run this program on our Windows system, for example, `long` is 4 bytes and `long long` is 8 bytes, whereas on our Mac, they're both 8 bytes. In this example¹³, lines 7–15 implicitly initialize each variable to 0 using a C++11 empty initializer list, `{}`.

¹³. Line 16 uses `const` rather than `constexpr` to prevent a type mismatch compilation error. The name of the built-in array of `ints` (line 15) decays to a `const int*`, so we must declare `ptr` with that type.

[Click here to view code image](#)

```
1 // fig07_11.cpp
2 // sizeof operator used to determine stand-
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     constexpr char c{}; // variable of type
8     constexpr short s{}; // variable of type
9     constexpr int i{}; // variable of type
10    constexpr long l{}; // variable of type
11    constexpr long long ll{}; // variable o
```

```

12     constexpr float f{}; // variable of type float
13     constexpr double d{}; // variable of type double
14     constexpr long double ld{}; // variable of type long double
15     constexpr int array[20]{ }; // built-in array
16     const int* const ptr{array}; // variable of type const int*
17
18     cout << "sizeof c = " << sizeof c
19             << "\nsizeof(char) = " << sizeof(char)
20             << "\nsizeof s = " << sizeof s
21             << "\nsizeof(short) = " << sizeof(short)
22             << "\nsizeof i = " << sizeof i
23             << "\nsizeof(int) = " << sizeof(int)
24             << "\nsizeof l = " << sizeof l
25             << "\nsizeof(long) = " << sizeof(long)
26             << "\nsizeof ll = " << sizeof ll
27             << "\nsizeof(long long) = " << sizeof(long long)
28             << "\nsizeof f = " << sizeof f
29             << "\nsizeof(float) = " << sizeof(float)
30             << "\nsizeof d = " << sizeof d
31             << "\nsizeof(double) = " << sizeof(double)
32             << "\nsizeof ld = " << sizeof ld
33             << "\nsizeof(long double) = " << sizeof(long double)
34             << "\nsizeof array = " << sizeof(array)
35             << "\nsizeof ptr = " << sizeof(ptr) << endl;
36 }

```

< >

sizeof c = 1	sizeof(char) = 1
sizeof s = 2	sizeof(short) = 2
sizeof i = 4	sizeof(int) = 4
sizeof l = 8	sizeof(long) = 8
sizeof ll = 8	sizeof(long long) = 8
sizeof f = 4	sizeof(float) = 4
sizeof d = 8	sizeof(double) = 8
sizeof ld = 16	sizeof(long double) = 16
sizeof array = 80	
sizeof ptr = 8	

Fig. 7.11 `sizeof` operator used to determine standard data type sizes.

The number of bytes used to store a particular data type may vary among systems and compilers. When writing programs that depend on data type sizes, always use `sizeof` to determine the number of bytes used to store the data types.

Operator `sizeof` can be applied to any expression or type name. When applied to a variable name (which is not a built-in array's name) or other expression, the number of bytes used to store the corresponding type is returned. The parentheses used with `sizeof` are required only if a type name (e.g., `int`) is supplied as its operand. The parentheses used with `sizeof` are not required when `sizeof`'s operand is an expression. Remember that `sizeof` is a compile-time operator, so its operand is not evaluated at runtime.

7.9 Pointer Expressions and Pointer Arithmetic

C++ enables **pointer arithmetic**—arithmetic operations that may be performed on pointers. This section describes the operators that have pointer operands and how these operators are used with pointers.

CG Pointer arithmetic is appropriate only for pointers that point to built-in array elements. You're likely to encounter pointer arithmetic in legacy code. However, **the C++ Core Guidelines indicate that a pointer should refer only to a single object (not an array),¹⁴ and that you should not use pointer arithmetic because it's highly error prone.¹⁵** If you need to process built-in arrays, use C++20 spans instead ([Section 7.10](#)).

14. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr>.

15. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-bounds>.

Valid pointer arithmetic operations are:

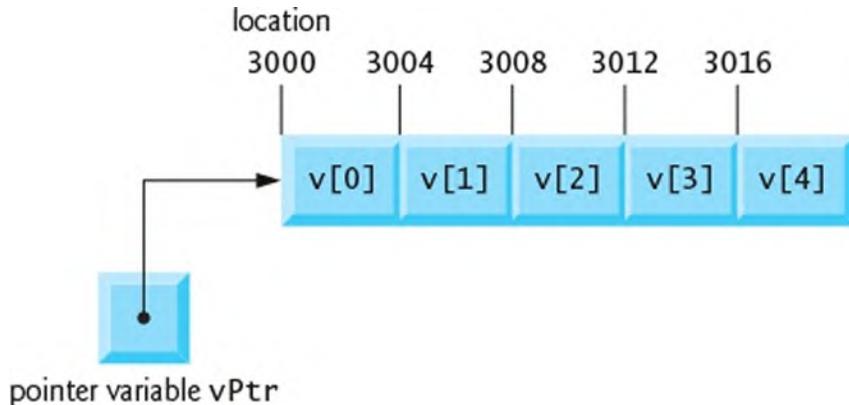
- incrementing `(++)` or decrementing `(--)`,

- adding an integer to a pointer (+ or +=) or subtracting an integer from a pointer (- or -=), and
- subtracting one pointer from another of the same type

Subtracting pointers is appropriate only for two pointers that point to elements of the same built-in array.

Most computers today have four-byte (32-bit) or eight-byte (64-bit) integers, though some of the billions of resource-constrained Internet of Things (IoT) devices are built using 8-bit or 16-bit hardware. Integer sizes typically are based on the hardware architecture, so such hardware might use one- or two-byte integers, respectively. The results of pointer arithmetic depend on the size of the memory objects a pointer points to, so pointer arithmetic is machine-dependent.

Assume that `int v[5]` has been declared and that its first element is at memory location 3000. Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000). The following diagram illustrates this situation for a machine with four-byte integers:



Variable `vPtr` can be initialized to point to `v` with either of the following statements (because a built-in array's name implicitly converts to the address of its zeroth element):

```
int* vPtr{v};  
int* vPtr{&v[0]};
```

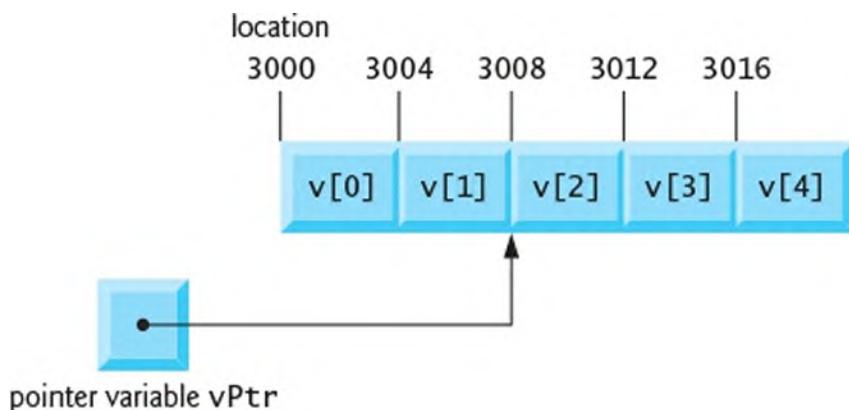
7.9.1 Adding Integers to and Subtracting Integers from Pointers

In conventional arithmetic, the addition $3000 + 2$ yields the value 3002.

This is normally not the case with pointer arithmetic. Adding an integer to or subtracting an integer from a pointer increments or decrements the pointer by that integer times the size of the type to which the pointer refers. The number of bytes depends on the memory object's data type. For example, the statement

```
vPtr += 2;
```

would produce 3008 (from the calculation $3000 + 2 * 4$), assuming that an `int` is stored in four bytes of memory. In the built-in array `v`, `vPtr` would now point to `v[2]` as in the diagram below:



If `vPtr` had been incremented to 3016, which points to `v[4]`, the statement

```
vPtr -= 4;
```

would set `vPtr` back to 3000—the beginning of the built-in array. If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used. Each of the statements

```
++vPtr;  
vPtr++;
```

increments the pointer to point to the built-in array's next element. Each of the statements

```
--vPtr;  
vPtr--;
```

decrements the pointer to point to the built-in array's previous element.

CG There's no bounds checking on pointer arithmetic, so the C++ Core

Guidelines recommend using `std::spans` instead, which we demonstrate in Section 7.10. You must ensure that every pointer arithmetic operation that adds an integer to or subtracts an integer from a pointer results in a pointer that references an element within the built-in array's bounds. As you'll see, `std::spans` have bounds checking, which helps you avoid errors.

7.9.2 Subtracting One Pointer from Another

Pointer variables pointing to the same built-in array may be subtracted from one another. For example, if `vPtr` contains the address 3000 and `v2Ptr` contains the address 3008, the statement

```
x = v2Ptr - vPtr;
```

would assign to `x` the number of built-in array elements from `vPtr` to `v2Ptr`—in this case, 2. Pointer arithmetic is meaningful only on a pointer that points to a built-in array. We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of a built-in array. Subtracting or comparing two pointers that do not refer to elements of the same built-in array is a logic error.

7.9.3 Pointer Assignment

A pointer can be assigned to another pointer if both pointers are of the same type.¹⁶ The exception to this rule is the **pointer to void** (i.e., `void*`), which is a pointer capable of representing any pointer type. Any pointer to a fundamental type or class type can be assigned to a pointer of type `void*` without casting. However, a pointer of type `void*` cannot be assigned directly to a pointer of another type—the pointer of type `void*` must first be cast to the proper pointer type (generally via a `reinterpret_cast`; discussed in Section 9.8).

¹⁶. Of course, `const` pointers cannot be modified.

7.9.4 Cannot Dereference a `void*`

A `void*` pointer cannot be dereferenced. For example, the compiler “knows” that an `int*` points to four bytes of memory on a machine with four-byte integers. Dereferencing an `int*` creates an *lvalue* that is an alias

for the `int`'s four bytes in memory. A `void*`, however, simply contains a memory address for an unknown data type. You cannot dereference a `void*` because the compiler does not know the type of the data to which the pointer refers and thus not the number of bytes.

The allowed operations on `void*` pointers are:

- comparing `void*` pointers with other pointers,
- casting `void*` pointers to other pointer types and
- assigning addresses to `void*` pointers.

All other operations on `void*` pointers are compilation errors.

7.9.5 Comparing Pointers

Pointers can be compared using equality and relational operators. Relational comparisons using are meaningless unless the pointers point to elements of the same built-in array. Pointer comparisons compare the addresses stored in the pointers. Comparing two pointers pointing to the same built-in array could show, for example, that one pointer points to a higher-numbered element than the other. A common use of pointer comparison is determining whether a pointer has the value `nullptr` (i.e., a pointer to nothing).

7.10 Objects Natural Case Study: C++20 spans—Views of Contiguous Container Elements

We now continue our objects natural approach by taking C++20 `span` objects for a spin. A `span` (header ``) enables programs to view contiguous elements of a container, such as a built-in array, a `std::array` or a `std::vector`. A `span` is a “view” into a container—it “sees” the container’s contents, but does not have its own copy of the container’s data.

CG Earlier, we discussed how C++ built-in arrays decay to pointers when passed to functions. In particular, the function’s parameter loses the size information that was provided when you declared the array. You saw this in our `sizeof` demonstration in Fig. 7.10. The C++ Core Guidelines recommend passing built-in arrays to functions as `spans`¹⁷, which represent both a pointer to the array’s first element and the array’s size.

Figure 7.12 demonstrates some key span capabilities.

17. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ap>.

[Click here to view code image](#)

```
1 // fig07_12.cpp
2 // C++20 spans: Creating views into containers
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 #include <span>
7 #include <vector>
8 using namespace std;
9
```



Fig. 7.12 C++20 spans: Creating views into containers.

Function `displayArray`

 **Security CG** Passing a built-in array to a function typically requires both the array's name and the array's size. The parameter `items` (line 12), though declared with `[]`, is simply a pointer to an `int`—the pointer does not “know” how many elements the function’s argument contains. There are various problems with this approach. For instance, the code that calls `displayArray` could pass the wrong value for `size`. In this case, the function might not process all of `items`’ elements, or the function might access an element outside `items`’ bounds—a logic error and a potential security issue. In addition, we previously discussed the disadvantages of external iteration, as used in lines 13–15. The C++ Core Guidelines checker in Visual Studio issues several warnings about `displayArray` and passing built-in arrays to functions. We include function `displayArray` in this example only for comparison with passing `spans` in function `displaySpan`, which is the recommended approach.

[Click here to view code image](#)

```
10 // items parameter is treated as a const int*
11 // know how to iterate over items with counter
12 void displayArray(const int items[], size_t si
13     for (size_t i{0}; i < size; ++i) {
14         cout << items[i] << " ";
15     }
16 }
17
```



Function `displaySpan`

CG The C++ Core Guidelines indicate that a pointer should point only to one object, not an array¹⁸ and that functions like `displayArray`, which receive a pointer and a size, are error-prone.¹⁹ To fix these issues, you should pass arrays to functions using spans, as in `displaySpan` (lines 20–24), which receives a span containing `const` `ints` because the function does not need to modify the data to display it.

18. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr>.

19. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array>.

[Click here to view code image](#)

```
18 // span parameter contains both the location of
19 // and the number of elements, so we can iterate
20 void displaySpan(span<const int> items) {
21     for (const auto& item : items) { // spans are
22         cout << item << " ";
23     }
24 }
25
```



CG ⚡ PERF A span encapsulates both a pointer and a count of the number of contiguous elements. When you pass a built-in array to `displaySpan`, C++ implicitly creates a span containing a pointer to the array's first element and the array's size, which the compiler can determine from the array's declaration. This span is a view of the data in the original array that you pass as an argument. The C++ Core Guidelines indicate that you can pass a span by value because it's just as efficient as passing the pointer and size separately²⁰, as we did in `displayArray`.

20. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-range>.

Security A span has many capabilities similar to arrays and vectors, such as iteration via the range-based `for` statement. Because a span is created based on the array's original size as determined by the compiler, the range-based `for` guarantees that we cannot access an element outside the bounds of the array that the span views, thus fixing the various problems associated with `displayArray` and helping prevent security issues like buffer overflows.

Function `times2`

A span is a view into an existing container, so changing the span's elements changes the container's original data. Function `times2` multiplies every item in its `span<int>` by 2. Note that we use a non-const reference to modify each element that the span views.

[Click here to view code image](#)

```
26 // spans can be used to modify elements in the
27 void times2(span<int> items) {
28     for (int& item : items) {
29         item *= 2;
30     }
31 }
32
```



Passing an Array to a Function to Display the Contents

Lines 34–36 create the `int` built-in array `values1`, the `std::array` `values2` and the `std::vector` `values3`. Each has five elements and stores its elements contiguously in memory. Line 41 calls `displayArray` to display `values1`'s contents. The `displayArray` function's first parameter is a pointer to an `int`, so we cannot use a `std::array`'s or `std::vector`'s name to pass these objects to `displayArray`.

[Click here to view code image](#)

```
33 int main() {
34     int values1[5]{1, 2, 3, 4, 5};
35     array<int, 5> values2{6, 7, 8, 9, 10};
36     vector<int> values3{11, 12, 13, 14, 15};
37
38     // must specify size because the compiler tre
39     // parameter as a pointer to the first elemen
40     cout << "values1 via displayArray: ";
41     displayArray(values1, 5);
42 }
```



```
values1 via displayArray: 1 2 3 4 5
```

Implicitly Creating spans and Passing Them to Functions

Line 46 calls `displaySpan` with `values1` as an argument. The function's parameter was declared as

```
span<const int>
```

so C++ creates a span containing a `const int*` that points to the array's first element and the array's size, which the compiler gets from the declaration of `values1` (line 34). Because spans can view any contiguous sequence of elements, you may also pass a `std::array` or `std::vector` of `int` to `displaySpan`, and C++ will create an

appropriate span representing a pointer to the container's first element and the container's size. This makes function `displaySpan` more flexible than `displayArray`, which could receive only the built-in array in this example.

[Click here to view code image](#)

```
43     // compiler knows values' size and automatically
44     // representing &values1[0] and the array's 1
45     cout << "\nvalues1 via displaySpan: ";
46     displaySpan(values1);
47
48     // compiler also can create spans from std::array
49     cout << "\nvalues2 via displaySpan: ";
50     displaySpan(values2);
51     cout << "\nvalues3 via displaySpan: ";
52     displaySpan(values3);
53
```

```
< [REDACTED] >
values1    via    displayArray: 1 2 3 4 5
values1    via    displaySpan: 1 2 3 4 5
values2    via    displaySpan: 6 7 8 9 10
values3    via    displaySpan: 11 12 13 14 15
```

Changing a span's Elements Modifies the Original Data

As we mentioned, function `times2` multiplies its span's elements by 2. Line 55 calls `times2` with `values1` as an argument. The function's parameter was declared as

```
span<int>
```

so C++ creates a span containing an `int*` that points to the array's first element and the array's size, which the compiler gets from the declaration of `values1` (line 34). To prove that `times2` modified the original array's data, line 57 displays `values1`'s updated values. Like `displaySpan`, `times2` can be called with this program's `std::array` or

`std::vector` as well.

[Click here to view code image](#)

```
54     // changing a span's contents modifies the o
55     times2(values1);
56     cout << "\n\nvalues1 after times2 modifies i
57     displaySpan(values1);
58
```

```
< [ ] >
values1 after times2 modifies its span argument: 2
```

Manually Creating a Span and Interacting with It

You can explicitly create spans and interact with them. Line 60 creates a `span<int>` that views the data in `values1`. Lines 61–62 demonstrate the span's `front` and `back` member functions, which return the first and last element of the view, and thus, the first and last element of the built-in array `values1`, respectively.

[Click here to view code image](#)

```
59     // spans have various array-and-vector-like
60     span<int> mySpan{values1};
61     cout << "\n\nmySpan's first element: " << my
62             << "\nmySpan's last element: " << mySpan.
63
```

```
< [ ] >
mySpan's first element: 2
mySpan's last element: 10
```

CG An essential philosophy of the C++ Core Guidelines is to “prefer compile-time checking to runtime checking.”²¹ This enables the compiler

to find and report errors at compile-time, rather than you having to write code to help prevent runtime errors. In line 60, the compiler determines the span's size (5) from the `values1` declaration in line 34. You can state the span's size, as in

21. C++ Core Guidelines. Accessed June 14, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-compile-time>.

```
span<int, 5> mySpan{values1};
```

In this case, the compiler ensures that the span's declared size matches `values1`'s size; otherwise, a compilation error occurs.

Using a span with the Standard Library's accumulate Algorithm

As you've seen in this example, spans are iterable. This means you also can use the `begin` and `end` functions with spans to pass them to C++ standard library algorithms, such as `accumulate` (line 66) or `sort`. We cover standard library algorithms in depth in [Chapter 18](#).

[Click here to view code image](#)

```
64 // spans can be used with standard library al
65 cout << "\n\nSum of mySpan's elements: "
66     << accumulate(begin(mySpan), end(mySpan),
67
```



```
Sum of mySpan's elements: 30
```

Creating Subviews

Sometimes, you might want to process subsets of the data a span views. A span's `first`, `last` and `subspan` member functions create subviews, which are themselves views. Lines 70 and 72 use `first` and `last` to get spans representing the first three and last three elements of `values1`, respectively. Line 74 uses `subspan` to get a span that views the 3 elements starting from index 1. In each case, we pass the subview's span to `displaySpan` to confirm what the span represents.

[Click here to view code image](#)

```
68 // spans can be used to create subviews of a
69 cout << "\n\nFirst three elements of mySpan:
70 displaySpan(mySpan.first(3));
71 cout << "\nLast three elements of mySpan: ";
72 displaySpan(mySpan.last(3));
73 cout << "\nMiddle three elements of mySpan: "
74 displaySpan(mySpan.subspan(1, 3));
75
```

```
< ----- >
First three elements of mySpan: 2 4 6
Last three elements of mySpan: 6 8 10
Middle three elements of mySpan: 4 6 8
```

Changing a Subview's Elements Modifies the Original Data

A subview of non-const data can modify that data. Line 77 passes to function times2 a span that views the 3 elements starting from index 1 of values1. Line 79 displays the updated values1 elements to confirm the results.

[Click here to view code image](#)

```
76 // changing a subview's contents modifies the
77 times2(mySpan.subspan(1, 3));
78 cout << "\n\nvalues1 after modifying middle t
79 displaySpan(values1);
80
```

```
< ----- >
values1 after modifying middle three elements via s
```

Accessing a View's Elements Via the [] Operator

Like built-in arrays, `std::arrays` and `std::vectors`, you can access and modify span elements via the `[]` operator. Line 82 displays the element at index 2. Line 85 attempts to access an element that does not exist. On the Microsoft Visual C++ compiler, this results in an exception that displays the message,²² "Expression: span index out of range".

22. At the time of this writing, the draft C++20 standard document makes no mention of the `[]` operator throwing an exception. Neither GNU C++ nor the Apple Clang C++ throw exceptions on line 85. They simply display whatever is in that memory location.

[Click here to view code image](#)

```
81      // access a span element via []
82      cout << "\n\nThe element at index 2 is: " <<
83
84      // attempt to access an element outside the
85      cout << "\n\nThe element at index 10 is: " <
86 }
```

< >

The element at index 2 is: 12

7.11 A Brief Intro to Pointer-Based Strings

We've already used the C++ Standard Library `string` class to represent strings as full-fledged objects. Chapter 8 presents class `std::string` in detail. This section introduces C-style, pointer-based strings (as defined by the C programming language). Here, we'll refer to these as **C-strings** or strings and use `std::string` when referring to the C++ standard library's `string` class.

 **Security** `std::string` is preferred because it eliminates many of the security problems and bugs that can be caused by manipulating C-strings. However, there are some cases in which C-strings are required, such as reading in command-line arguments. Also, if you work with legacy C and C++ programs, you're likely to encounter pointer-based strings. We cover C-strings in detail in Appendix F.

Characters and Character Constants

Characters are the fundamental building blocks of C++ source programs. Every program is composed of characters that—when grouped meaningfully—are interpreted by the compiler as instructions and data used to accomplish a task. A program may contain **character constants**, each of which is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, 'z' represents the integer value of z (122 in the ASCII character set; see Appendix B), and '\n' represents the integer value of newline (10 in the ASCII character set).

Pointer-Based Strings

A C-string (also called a pointer-based string) is a built-in array of characters ending with a **null character** ('\0'), which marks where the string terminates in memory. A C-string is accessed via a pointer to its first character (no matter how long the string is). The result of `sizeof` for a string literal (which is a C-string) is the length of the string, including the terminating null character.

String Literals as Initializers

A string literal may be used as an initializer in the declaration of either a built-in array of `char`s or a variable of type `const char*`. The declarations

```
char color[] {"blue"};
const char* colorPtr {"blue"};
```

each initialize a variable to the string "blue". The first declaration creates a five-element built-in array `color` containing the characters 'b', 'l', 'u', 'e' and '\0'. The second declaration creates pointer variable `colorPtr` that points to the letter b in the string "blue" (which ends in '\0') somewhere in memory. The first declaration above also may be implemented using an initializer list of individual characters, as in:

[Click here to view code image](#)

```
char color[] {'b', 'l', 'u', 'e', '\0'};
```

String literals exist for the duration of the program. They may be shared if the

same string literal is referenced from multiple locations in a program. String literals are immutable—they cannot be modified.

Problems with C-Strings

Not allocating sufficient space in a built-in array of `chars` to store the null character that terminates a string is a logic error. Creating or using a C-string that does not contain a terminating null character can lead to logic errors.

 **Security** When storing a string of characters in a built-in array of `chars`, be sure that the built-in array is large enough to hold the largest string that will be stored. C++ allows strings of any length. If a string is longer than the built-in array of `chars` in which it's to be stored, characters beyond the end of the built-in array will overwrite subsequent memory locations. This could lead to logic errors, program crashes or security breaches.

Displaying C-Strings

A built-in array of `chars` representing a null-terminated string can be output with `cout` and `<<`. The statement

```
cout << sentence;
```

displays the built-in array `sentence`. `cout` does not care how large the built-in array of `chars` is. The characters are output until a terminating null character is encountered; the null character is not displayed. `cin` and `cout` assume that built-in arrays of `chars` should be processed as strings terminated by null characters. `cin` and `cout` do not provide similar input and output processing capabilities for other built-in array types.

7.11.1 Command-Line Arguments

There are cases in which built-in arrays and C-strings must be used, such as processing a program's **command-line arguments**, which are often passed to applications to specify configuration options, file names to process and more.

You supply command-line arguments to a program by placing them after the program's name when executing it from the command line. Such arguments typically pass options to a program. For example, on a Windows system, the command

```
dir /p
```

uses the `/p` argument to list the contents of the current directory, pausing after each screen of information. Similarly, on Linux or macOS, the following command uses the `-la` argument to list the contents of the current directory with details about each file and directory:

```
ls -la
```

Command-line arguments are passed into a C++ program as C-strings, and the application name is treated as the first command line argument. To use the arguments as `std::strings` or other data types (`int`, `double`, etc.), you must convert the arguments to those types. [Figure 7.13](#) displays the number of command-line arguments passed to the program, then displays each argument on a separate line of output.

[Click here to view code image](#)

```
1 // fig07_13.cpp
2 // Reading in command-line arguments.
3 #include <iostream>
4 using namespace std;
5
6 int main(int argc, char* argv[]) {
7     cout << "There were " << argc << " command-line arguments." << endl;
8     for (int i{0}; i < argc; ++i) {
9         cout << argv[i] << endl;
10    }
11 }
```



fig07_13 Amanda Green 97

There were 4 command-line arguments
fig07_13
Amanda
Green
97

Fig. 7.13 Reading in command-line arguments.

To receive command-line arguments, declare `main` with two parameters (line 6), which by convention are named `argc` and `argv`, respectively. The first is an `int` representing the number of arguments. The second is a `char*` built-in array. The first element of the array is a C-string for the application name. The remaining elements are C-strings for the other command-line arguments.

The command

```
fig07_13 Amanda Green 97
```

passes "Amanda", "Green" and "97" to the application `fig07_13` (on macOS and Linux you'd run this program with `./fig07_13`). Command-line arguments are separated by white space, *not* commas. When this command executes, `fig07_13`'s `main` function receives the argument count 4 and a four-element array of C-strings:

- `argv[0]` contains the application's name "`fig07_13`" (or `./fig07_13` on macOS or Linux), and
- `argv[1]` through `argv[3]` contain "Amanda", "Green" and "97", respectively.

You determine how to use these arguments in your program.

7.11.2 Revisiting C++20's `to_array` Function

20 Section 7.6 demonstrated converting built-in arrays to `std::arrays` with `to_array`. Figure 7.14 shows another purpose of `to_array`. We use the same lambda expression (lines 9–13) as in Fig. 7.6 to display the `std::array` contents after each `to_array` call.

[Click here to view code image](#)

```
1 // fig07_14.cpp
2 // C++20: Creating std::arrays from string
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
```

```
7     int main() {
8         // lambda to display a collection of items
9         const auto display = [] (const auto& items) {
10             for (const auto& item : items) {
11                 cout << item << " ";
12             }
13         };
14     }
```



Fig. 7.14 C++20: Creating `std::array`s from string literals with `to_array`.

Initializing a `std::array` from a String Literal Creates a One-Element array

Function `to_array` fixes an issue with initializing a `std::array` from a string literal. Rather than creating a `std::array` of the individual characters in the string literal, line 17 creates a one-element array containing a `const char*` pointing to the C-string "abc".

[Click here to view code image](#)

```
15     // initializing an array with a string liter
16     // creates a one-element array<const char*>
17     const auto array1 = array{"abc"};
18     cout << "\n\narray1.size() = " << array1.size();
19     display(array1); // use lambda to display co
20
```

```
< ----- >
array1.size() = 1
array1: abc
```

Passing a String Literal to `to_array` Creates a `std::array` of `char`

On the other hand, passing a string literal to `to_array` (line 22) creates a `std::array` of `char`s containing elements for each character and the

terminating null character. Line 23 confirms that the array's size is 6. Line 24 confirms the array's contents. The null character does not have a visual representation, so it does not appear in the output.

[Click here to view code image](#)

```
21     // creating std::array of characters from a
22     const auto array2 = to_array("C++20");
23     cout << "\n\narray2.size() = " << array2.size()
24     display(array2); // use lambda to display co
25
26     cout << endl;
27 }
```

```
< >
array2.size() = 6
array2: C + + 2 0
```

7.12 Looking Ahead to Other Pointer Topics

In later chapters, we'll introduce additional pointer topics:

- In [Chapter 13](#), Object-Oriented Programming: Polymorphism, we'll use pointers with class objects to show that the “runtime polymorphic processing” associated with object-oriented programming can be performed with references or pointers—you should favor references.
- In [Chapter 14](#), Operator Overloading, we introduce dynamic memory management with pointers, which allows you at execution time to create and destroy objects as needed. Improperly managing this process is a source of subtle errors, such as “memory leaks.” We'll show how “smart pointers” can automatically manage memory and other resources that should be returned to the operating system when they're no longer needed.
- In [Chapter 18](#), Standard Library Algorithms, we show that a function's name is also a pointer to its implementation in memory, and that functions can be passed into other functions via function pointers—

exactly as lambda expressions are.

7.13 Wrap-Up

This chapter discussed pointers, built-in pointer-based arrays and pointer-based strings (C-strings). **We pointed out Modern C++ guidelines that recommend avoiding most pointers—preferring references to pointers, `std::array`²³ and `std::vector` objects to built-in arrays, and `std::string` objects to C-strings.**

²³. We pronounce “`std::`” as “standard,” so throughout this chapter we say “a `std::array`” rather than “an `std::array`,” which assumes “`std::`” is pronounced as its individual letters s, t and d.

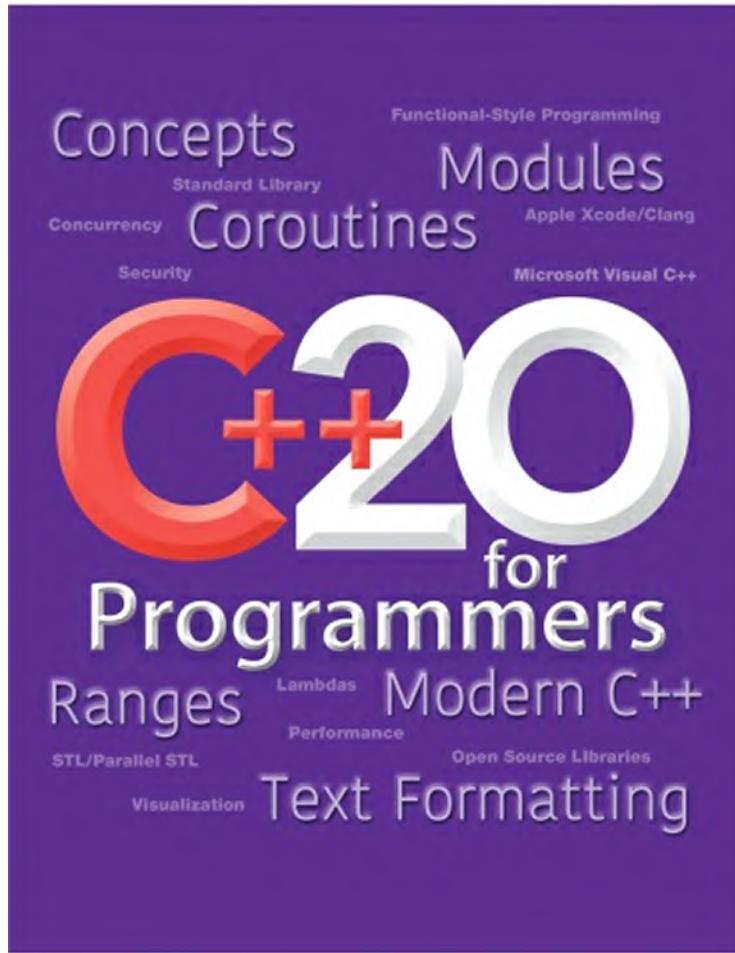
We declared and initialized pointers and demonstrated the pointer operators & and *. We showed that pointers enable pass-by-reference, but you should generally prefer references for that purpose. We used built-in, pointer-based arrays and showed their intimate relationship with pointers.

We discussed various combinations of `const` with pointers and the data they point to and used the `sizeof` operator to determine the number of bytes that store values of particular fundamental types and pointers. We demonstrated pointer expressions and pointer arithmetic.

We briefly discussed C-strings then showed how to process command-line arguments—a simple task for which C++ still requires you to use both pointer-based C-strings and pointer-based arrays.

As a reminder, the key takeaway from reading this chapter is that you should avoid using pointers, pointer-based arrays and pointer-based strings whenever possible. For programs that still use pointer-based arrays, you can use C++20’s `to_array` function to convert built-in arrays to `std::arrays` and C++20’s `spans` as a safer way to process built-in pointer-based arrays. In the next chapter, we discuss typical string-manipulation operations provided by `std::string` and introduce file-processing capabilities.

Chapter 8. `strings`, `string_views`, Text Files, CSV Files and Regex



Objectives

In this chapter, you'll:

- Determine `string` characteristics.
- Find, replace and insert characters in `strings`.
- Use C++11 numeric conversion functions.
- See the C++20 update to how `string` member function `reserve` modifies `string` capacity.

- Use C++17 `string_views` for lightweight views of contiguous characters.
 - Write and read sequential files.
 - Perform input from and output to `strings` in memory.
 - Do an objects-natural case study using an object of an open-source-library class to read and process data about the Titanic disaster from a CSV (comma-separated values) file.
 - Do an objects-natural case study using C++11 regular expressions (`regex`) to search strings for patterns, validate data and replace substrings.
-

Outline

- 8.1 Introduction
- 8.2 `string` Assignment and Concatenation
- 8.3 Comparing `strings`
- 8.4 Substrings
- 8.5 Swapping `strings`
- 8.6 `string` Characteristics
 - 8.6.1 C++20 Update to `string` Member-Function `reserve`
- 8.7 Finding Substrings and Characters in a `string`
- 8.8 Replacing Characters in a `string`
- 8.9 Inserting Characters into a `string`
- 8.10 C++11 Numeric Conversions
- 8.11 C++17 `string_view`
- 8.12 Files and Streams
- 8.13 Creating a Sequential File
- 8.14 Reading Data from a Sequential File
- 8.15 C++14 Reading and Writing Quoted Text
- 8.16 Updating Sequential Files
- 8.17 String Stream Processing
- 8.18 Raw String Literals
- 8.19 Objects Natural Case Study: Reading and Analyzing a CSV File

Containing Titanic Disaster Data

8.19.1 Using `rapidcsv` to Read the Contents of a CSV File

8.19.2 Reading and Analyzing the Titanic Disaster Dataset

8.20 Objects Natural Case Study: Introduction to Regular Expressions

8.20.1 Matching Complete Strings to Patterns

8.20.2 Replacing Substrings

8.20.3 Searching for Matches

8.21 Wrap-Up

8.1 Introduction

17 This chapter discusses additional `std::string` features and introduces C++17 `string_views`, text file-processing, CSV file processing and regular expressions.

`std::strings`

20 We've been using `std::string` object since Chapter 2. Here, we introduce many more `std::string` manipulations, including assignment, comparisons, extracting substrings, searching for substrings, modifying `std::string` objects and converting `std::string` objects to numeric values. We also introduce a C++20 change to the mechanics of the `std::string` member function `reserve`.

C++17 `string_views`

17 We introduce C++17's `string_views`, which are read-only views of C-strings or `std::string` objects. Like `std::span`, a `string_view` does not own the data it views. You'll see that `string_views` have many similar capabilities to `std::strings`, making them appropriate for many cases in which you do not need modifiable strings.

Text Files

Data storage in memory is temporary. **Files** are used for **data persistence**—permanent retention of data. Computers store files on **secondary storage devices**, such as flash drives, and frequently today, in the cloud. In this

chapter, we explain how to build C++ programs that create, update and process sequential text files. We also show how to output data to and read data from a `std::string` in memory using `ostringsstreams` and `istringsstreams`.

Objects Natural Case Study: CSV Files and the Titanic Disaster Dataset

In this chapter's first objects-natural case study, we introduce the CSV (comma-separated values) file format. CSV is popular for datasets used in big data, data analytics and data science, and artificial intelligence applications like natural language processing, machine learning and deep learning.

DS One of the most commonly used datasets for data analytics and data science beginners is the Titanic disaster dataset. It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank on its maiden voyage April 14–15, 1912. We use a class from the open-source `rapidcsv` library to create an object that reads the Titanic dataset from a CSV file. Then, we view some of the data and perform some basic data analytics.

Objects Natural Case Study: Using Regular Expressions to Search Strings for Patterns, Validate Data and Replace Substrings

11 In this chapter's second objects-natural case study, we introduce regular expressions, which are particularly crucial in today's data-rich applications. We'll use C++11 `regex` objects to create regular expressions then use them with various functions in the `<regex>` header to match patterns in text. In earlier chapters, we mentioned the importance of validating user input in industrial-strength code. The `std::string`, string stream and regular expression capabilities presented in this chapter are frequently used to validate data.

8.2 **string Assignment and Concatenation**

Figure 8.1 demonstrates `std::string` assignment and concatenation.

[Click here to view code image](#)

```
1 // fig08_01.cpp
2 // Demonstrating string assignment and concatenation
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this is part of the standard library
6 using namespace std;
7
8 int main() {
```



Fig. 8.1 Demonstrating string assignment and concatenation.

String Assignment

Lines 9–11 create the strings `s1`, `s2` and `s3`. Line 13 uses the assignment operator to copy the contents of `s1` into `s2`. You also can specify two arguments, as in

[Click here to view code image](#)

```
string bar{8, '*'}; // string of 8 '*' characters
```

which creates a string containing eight '*' characters.

[Click here to view code image](#)

```
9     string s1{"cat"};
10    string s2; // initialized to the empty string
11    string s3; // initialized to the empty string
12
13    s2 = s1; // assign s1 to s2
14    s3.assign(s1); // assign s1 to s3
15    cout << fmt::format("s1: {}\ns2: {}\ns3: {}\n")
16
```



```
s1: cat
s2: cat
```

```
s3: cat
```

Line 14 uses member function **assign** to copy s1's contents into s3. This particular version of assign is equivalent to using the operator, but assign also has many overloads that enable you to assign characters to an existing string object. For details of each, see

https://en.cppreference.com/w/cpp/string/basic_string/assign

std::string also provides an overloaded version of member function assign that copies a specified number of characters, as in

[Click here to view code image](#)

```
target.assign(source, start, numberOfChars);
```

where source is the string to copy, start is the starting index, and numberOfChars is the number of characters to copy.

Accessing String Elements By Index

Lines 17 and 18 use the string member function **at** to assign 'r' to s2 at index 0 (forming "rat") and to assign 'r' to s3 at index 2 (forming "car").

[Click here to view code image](#)

```
17    s2.at(0) = 'r'; // modify s2
18    s3.at(2) = 'r'; // modify s3
19    cout << fmt::format("After modifications:\n{s2}
20
```

```
After modifications:
s2: rat
s3: car
```

You also can use the member function **at** to get the character at a specific index in a string. As with std::array and std::vector, a

`std::string`'s `at` member function performs range checking and throws an `out_of_range` exception if the index is not within the string's bounds. The subscript operator, `[]`, also is available for strings, but does not provide checked access. This is consistent with its use with `std::array` and `std::vector`. You also can iterate through the characters in a string using range-based `for` as in

```
for (char c : s3) {  
    cout << c;  
}
```

which ensures that you do not access any elements outside the string's bounds.

Accessing String Elements By Index

Line 22 initializes `s4` to the contents of `s1` followed by "apult". For `std::string`, the `+` operator denotes string concatenation. Line 23 uses member function `append` to concatenate `s1` and "acomb". Next, line 24 uses the overloaded addition assignment operator, `+=`, to concatenate `s3` and "pet". Then line 30 appends the string "comb" to empty string `s5`. The arguments are the `std::string` to retrieve characters from (`s1`), the starting index (4) and the number of characters to append (`s1.size() - 4`).

[Click here to view code image](#)

```
21   cout << "\n\nAfter concatenations:\n";  
22   string s4{s1 + "apult"}; // concatenation  
23   s1.append("acomb"); // create "catacomb"  
24   s3 += "pet"; // create "carpet" with overload  
25   cout << fmt::format("s1: {}\ns3: {}\ns4: {}\n  
26  
27   // append locations 4 through end of s1 to  
28   // create string "comb" (s5 was initially emp  
29   string s5; // initialized to the empty string  
30   s5.append(s1, 4, s1.size() - 4);  
31   cout << fmt::format("s5: {}", s5);  
32 }
```

After concatenations:

```
s1: catacomb  
s3: carpet  
s4: catapult  
s5: comb
```

8.3 Comparing strings

`std::string` provides member functions for comparing strings (Fig. 8.2). Throughout this example, we call function `displayResult` (lines 8–18) to display each comparison’s result. The program declares four strings (lines 21–24) and outputs each (line 26).

[Click here to view code image](#)

```
1 // fig08_02.cpp  
2 // Comparing strings.  
3 #include <iostream>  
4 #include <string>  
5 #include "fmt/format.h" // In C++20, this  
6 using namespace std;  
7  
8 void displayResult(const string& s, int re  
9     if (result == 0) {  
10         cout << s + " == 0\n";  
11     }  
12     else if (result > 0) {  
13         cout << s + " > 0\n";  
14     }  
15     else { // result < 0  
16         cout << s + " < 0\n";  
17     }  
18 }  
19 }
```

```
20     int main() {
21         const string s1{"Testing the comparison
22         const string s2{"Hello"};
23         const string s3{"stinger"};
24         const string s4{s2}; // "Hello"
25
26         cout << fmt::format("s1: {}\\ns2: {}\\ns3:
27
```

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
s4: Hello
```

Fig. 8.2 Comparing strings.

Comparing Strings with the Relational and Equality Operators

Strings may be compared with the relational and equality operators—each returns a `bool`. Comparisons are performed **lexicographically**—that is, based on the integer values of each character (see [Appendix B, ASCII Character Set](#)). Line 29 tests whether `s1` is greater than `s4` using the overloaded `>` operator. In this case, `s1` starts with a capital T, and `s4` starts with a capital H. So, `s1` is greater than `s4` because T has a higher numeric value than H.

[Click here to view code image](#)

```
28     // comparing s1 and s4
29     if (s1 > s4) {
30         cout << "\\n\\ns1 > s4\\n";
31     }
32
```

```
s1 > s4
```

Comparing Strings with Member Function `compare`

Line 34 compares `s1` to `s2` using `std::string` member function **compare**. This member function returns 0 if the strings are equal, a positive number if `s1` is **lexicographically** greater than `s2` or a negative number if `s1` is lexicographically less than `s2`. Because a string starting with 'T' is considered lexicographically greater than a string starting with 'H', result is assigned a value greater than 0, as confirmed by the output.

[Click here to view code image](#)

```
33 // comparing s1 and s2
34 displayResult("s1.compare(s2)", s1.compare(s2)
35
36 // comparing s1 (elements 2-5) and s3 (eleme
37 displayResult("s1.compare(2, 5, s3, 0, 5)", 
38     s1.compare(2, 5, s3, 0, 5));
39
40 // comparing s2 and s4
41 displayResult("s4.compare(0, s2.size(), s2)", 
42     s4.compare(0, s2.size(), s2));
43
44 // comparing s2 and s4
45 displayResult("s2.compare(0, 3, s4)", s2.comp
46 }
```

```
s1.compare(s2) > 0
s1.compare(2, 5, s3, 0, 5) == 0
s4.compare(0, s2.size(), s2) == 0
s2.compare(0, 3, s4) < 0
```

The call to `compare` in line 38 compares portions of `s1` and `s3` using an overloaded version of member function `compare`. The first two arguments (2 and 5) specify the starting index and length of the portion of `s1` ("string") to compare with `s3`. The third argument is the comparison string. The last two arguments (0 and 5) are the starting index and length

of the portion of the comparison string being compared (also "sting"). The two pieces being compared here are identical, so `compare` returns 0 as confirmed in the output.

Line 42 uses another overloaded version of function `compare` to compare `s4` and `s2`. The first two arguments are the starting index and length. The last argument is the comparison string. The pieces of `s4` and `s2` being compared are identical, so `compare` returns 0.

Line 45 compares the first 3 characters in `s2` to `s4`. Because "Hel" begins with the same first three letters as "Hello" but has fewer letters overall, "Hel" is considered less than "Hello" and `compare` returns a value less than zero.

8.4 Substrings

`std::string`'s member function `substr` (Fig. 8.3) returns a substring from a string. The result is a new `string` object that's copied from the source `string`. Line 9 uses member function `substr` to get a substring from `s` starting at index 3 and consisting of 4 characters.

[Click here to view code image](#)

```
1 // fig08_03.cpp
2 // Demonstrating string member function substr
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main() {
8     const string s{"airplane"};
9     cout << s.substr(3, 4) << endl; // retrieves
10 }
```

plan

Fig. 8.3 Demonstrating string member function substr.

8.5 Swapping strings

`std::string` provides member function `swap` for swapping strings. Figure 8.4 `swap` (line 14) to exchange the values of `first` and `second`.

[Click here to view code image](#)

```
1 // fig08_04.cpp
2 // Using the swap function to swap two strings
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this is part of the standard library
6 using namespace std;
7
8 int main() {
9     string first{"one"};
10    string second{"two"};
11
12    cout << fmt::format(
13        "Before swap:\nfirst: {};\nsecond: {}",
14        first.swap(second)); // swap strings
15    cout << fmt::format(
16        "\n\nAfter swap:\nfirst: {};\nsecond: {}",
17    }
```



```
Before swap:
first: one; second: two
```

```
After swap:
first: two; second: one
```

Fig. 8.4 Using the `swap` function to swap two strings.

8.6 string Characteristics

`std::string` provides member functions for gathering information about a string's size, capacity, maximum length and other characteristics:

- A string's size is the number of characters currently stored in the string.
- A string's **capacity** is the number of characters that can be stored in the string before it must allocate more memory to store additional characters. A string performs memory allocation for you behind the scenes. The capacity of a string must be at least equal to the current size of the string, though it can be greater. The exact capacity of a string depends on the implementation.
- The **maximum size** is the largest possible size a string can have. If this value is exceeded, a `length_error` exception is thrown.

Figure 8.5 demonstrates string member functions for determining these characteristics. Function `printStatistics` (lines 9–12) receives a string and displays its capacity (using member function `capacity`), maximum size (using member function `max_size`), size (using member function `size`), and whether the string is empty (using member function `empty`).

[Click here to view code image](#)

```
1 // fig08_05.cpp
2 // Printing string characteristics.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this
6 using namespace std;
7
8 // display string statistics
9 void printStatistics(const string& s) {
10     cout << fmt::format("capacity: {}\nmax
11             s.capacity(), s.max_size(), |
```

```
12      }
13
14  int main() {
```



Fig. 8.5 Printing string characteristics.

The program declares empty string `string1` (line 15) and passes it to function `printStatistics` (line 18). The initial call to `printStatistics` indicates that `string1`'s initial size is 0—it contains no characters. Recall that the size and length are always identical. On Visual C++, the maximum size (shown in the output) is 2,147,483,647. On GNU C++, the maximum is 9,223,372,036,854,775,807 and on Apple Clang it's 18,446,744,073,709,551,599. Object `string1` is an empty string, so function `empty` returns `true`.

[Click here to view code image](#)

```
15  string string1; // empty string
16
17  cout << "Statistics before input:\n";
18  printStatistics(string1);
19
```

```
Statistics before input:
capacity: 15
max size: 2147483647
size: 0
empty: true
```

Line 21 inputs a string (in this case, "tomato"). Line 24 calls `printStatistics` to output updated `string1` statistics. The size is now 6, and `string1` is no longer empty.

[Click here to view code image](#)

```
20     cout << "\n\nEnter a string: ";
21     cin >> string1; // delimited by whitespace
22     cout << "The string entered was: " << string1
23     cout << "\nStatistics after input:\n";
24     printStatistics(string1);
25
```

```
Enter a string: tomato
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 2147483647
size: 6
empty: false
```

Line 27 inputs another string (in this case, "soup") and stores it in `string1`, thereby replacing "tomato". Line 30 calls `printStatistics` to output updated `string1` statistics. Note that the length is now 4.

[Click here to view code image](#)

```
26     cout << "\n\nEnter a string: ";
27     cin >> string1; // delimited by whitespace
28     cout << "The string entered was: " << string1
29     cout << "\nStatistics after input:\n";
30     printStatistics(string1);
31
```

```
Enter a string: soup
The string entered was: soup
Statistics after input:
capacity: 15
max size: 2147483647
```

```
size: 4
empty: false
```

Line 33 uses `+=` to concatenate a 46-character string to `string1`. Line 36 calls `printStatistics` to output updated `string1` statistics. Because `string1`'s capacity was not large enough to accommodate the new string size, the capacity was automatically increased to 63 elements, and `string1`'s size is now 50.

[Click here to view code image](#)

```
32 // append 46 characters to string1
33 string1 += "1234567890abcdefghijklmnopqrstuvwxyz";
34 cout << "\n\nstring1 is now: " << string1;
35 cout << "\nStatistics after concatenation:\n";
36 printStatistics(string1);
37
```

```
< >
Statistics after resizing to add 10 characters:
capacity: 63
max_size: 2147483647
size: 60
empty: false
```

Line 38 uses member function `resize` to increase `string1`'s size by 10 characters. The additional elements are set to null characters. The `printStatistics` output shows that the capacity did not change, but the size is now 60.

[Click here to view code image](#)

```
38 string1.resize(string1.size() + 10); // add 1
39 cout << "\n\nStatistics after resizing to add
40 printStatistics(string1);
41 cout << endl;
```

```
42 }
```

```
< ━━━━ >  
Statistics after resizing to add 10 characters:  
capacity: 63  
max_size: 2147483647  
size: 60  
empty: false
```

8.6.1 C++20 Update to **string** Member-Function **reserve**

20 You can change the capacity of a **string** without changing its size by calling **string** member function **reserve**. If its integer argument is greater than the current capacity, the capacity is increased to greater than or equal to the argument value. As of C++20, if **reserve**'s argument is less than the current capacity, the capacity does not change. Before C++20, **reserve** optionally would reduce the capacity and, if the argument were smaller than the **string**'s size, optionally would reduce the capacity to match the size.

8.7 Finding Substrings and Characters in a **string**

std::string provides member functions for finding substrings and characters in a **string**. Figure 8.6 demonstrates the **find** functions. String **s** is declared and initialized in line 9.

[Click here to view code image](#)

```
1 // fig08_06.cpp
2 // Demonstrating the string find member fu:
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this is
6 using namespace std;
7
8 int main() {
```

```
9     const string s{"noon is 12pm; midnight"};
10    cout << "Original string: " << s;
11
```

```
Original string: noon is 12pm; midnight is not
```

Fig. 8.6 Demonstrating the `string` `find` member functions.

Member Functions `find` and `rfind`

Lines 13–14 attempt to find "is" in `s` using member functions `find` and `rfind`, which search from the beginning and end of `s`, respectively. If "is" is found, the index of the starting location of that string is returned. If the string is not found, the `string` `find`-related functions return the constant `string::npos` to indicate that a substring or character was not found in the `string`. The rest of the `find` functions presented in this section return the same type unless otherwise noted.

[Click here to view code image](#)

```
12 // find "is" from the beginning and end of s
13 cout << fmt::format("\ns.find(\"is\"): {}\\ns.\n"
14             s.find("is"), s.rfind("is"));
```

```
s.find("is"): 5
s.rfind("is"): 23
```

Member Function `find_first_of`

Line 17 uses member function `find_first_of` to locate the first occurrence in `s` of any character in "misop". The searching is done from the beginning of `s`. The character 'o' is found in element 1.

[Click here to view code image](#)

```
16 // find 'o' from beginning
17 int location = s.find_first_of("misop");
18 cout << fmt::format("\ns.find_first_of(\"miso
19                                     s.at(location), location);
20
```

```
s.find_first_of("misop") found o at 1
```

Member Function find_last_of

Line 22 uses member function **find_last_of** to find the last occurrence in *s* of any character in "misop". The searching is done from the end of *s*. The character 'o' is found in element 28.

[Click here to view code image](#)

```
21 // find 'o' from end
22 location = s.find_last_of("misop");
23 cout << fmt::format("\ns.find_last_of(\"misop
24                                     s.at(location), location);
25
```

```
s.find_last_of("misop") found o at 27
```

Member Function find_first_not_of

Line 27 uses member function **find_first_not_of** to find the first character from the beginning of *s* that is not contained in "noi spm", finding '1' in element 8. Line 33 uses member function **find_first_not_of** to find the first character not contained in "12noi spm". It searches from the beginning of *s* and finds ';' in element 12. Line 39 uses member function **find_first_not_of** to find the first character not contained in "noon is 12pm; midnight is not". In this case, the string being searched contains every character specified in the string

argument. Because a character was not found, `string::npos` (which has the value `-1` in this case) is returned.

[Click here to view code image](#)

```
26     // find '1' from beginning
27     location = s.find_first_not_of("noi spm");
28     cout << fmt::format(
29         "\ns.find_first_not_of(\"noi spm\""
30             s.at(location), location);
31
32     // find '.' at location 13
33     location = s.find_first_not_of("12noi spm");
34     cout << fmt::format(
35         "\ns.find_first_not_of(\"12noi spm"
36             s.at(location), location);
37
38     // search for characters not in "noon is 12pm"
39     location = s.find_first_not_of("noon is 12pm");
40     cout << fmt::format("\ns.find_first_not_of("
41             "\"noon is 12pm; midnight is not\""
42 }
```

< >

```
s.find_first_not_of("noi spm") found 1 at 8
s.find_first_not_of("12noi spm") found ; at 12
s.find_first_not_of("noon is 12pm; midnight is not"
```

< >

8.8 Replacing Characters in a `string`

Figure 8.7 demonstrates `string` member functions for replacing and erasing characters. Lines 10–14 declare and initialize `string` `string1`.

[Click here to view code image](#)

```
1 // fig08_07.cpp
2 // Demonstrating string member functions e.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this is
6 using namespace std;
7
8 int main() {
9 // compiler concatenates all parts into one
10 string string1{"The values in any left s-
11 " \nare less than the value in the"
12 " \nparent node and the values in"
13 " \nany right subtree are greater"
14 " \n\tthan the value in the parent node"};
15
16 cout << fmt::format("Original string:\n{
17
```

```
< >
```

Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Fig. 8.7 Demonstrating string member functions `erase` and `replace`.

Line 18 uses `string` member function `erase` to erase everything from (and including) the character in position 62 to the end of `string1`. Each newline character occupies one character in the string.

[Click here to view code image](#)

```
18     string1.erase(62); // remove from index 62 th
```

```
19     cout << fmt::format("string1 after erase:\n{}")
20
```

```
< ━━━━ >
string1 after erase:
The values in any left subtree
are less than the value in the
```

Lines 21–27 use `find` to locate each occurrence of the space character. Each space is then replaced with a period by a call to `string` member function `replace`. Function `replace` takes three arguments—the index of the character in the `string` at which replacement should begin, the number of characters to replace and the replacement string. Member function `find` returns `string::npos` when the search character is not found. In line 26, we add 1 to `position` to continue searching from the next character's location.

[Click here to view code image](#)

```
21     size_t position = string1.find(" "); // find
22
23     // replace all spaces with period
24     while (position != string::npos) {
25         string1.replace(position, 1, ".");
26         position = string1.find(" ", position + 1)
27     }
28
29     cout << fmt::format("After first replacement:
30
```

```
< ━━━━ >
After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the
```

Lines 31–38 use functions `find` and `replace` to find every period and

replace every period and its following character with two semicolons. The arguments passed to this version of `replace` are:

- the index of the element where the replace operation begins,
- the number of characters to replace,
- a replacement character string from which a substring is selected to use as replacement characters,
- the element in the character string where the replacement substring begins and
- the number of characters in the replacement character string to use.

[Click here to view code image](#)

```
31     position = string1.find("."); // find first p
32
33     // replace all periods with two semicolons
34     // NOTE: this will overwrite characters
35     while (position != string::npos) {
36         string1.replace(position, 2, "xxxxx;;yyy",
37         position = string1.find(".", position + 1)
38     }
39
40     cout << fmt::format("After second replacement
41 }
```

< >

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:
The;;values;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he

8.9 Inserting Characters into a string

`std::string` provides overloaded member functions for inserting characters into a `string` (Fig. 8.8). Line 14 uses `string` member function `insert` to insert "middle " before element 10 of `s1`. Line 15 uses `insert` to insert "xx" before `s2`'s element 3. The last two arguments specify the starting and last element of "xx" to insert. Using `string::npos` causes the entire string to be inserted.

[Click here to view code image](#)

```
1 // fig08_08.cpp
2 // Demonstrating std::string insert member
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this
6 using namespace std;
7
8 int main() {
9     string s1{"beginning end"};
10    string s2{"12345678"};
11
12    cout << fmt::format("Initial strings:\n");
13
14    s1.insert(10, "middle "); // insert "mi
15    s2.insert(3, "xx", 0, string::npos); // 
16
17    cout << fmt::format("Strings after inser
18 }
```

< >

```
Initial strings:
s1: beginning end
s2: 12345678
```

```
Strings after insert:
s1: beginning middle end
s2: 123xx45678
```

Fig. 8.8 Demonstrating `std::string insert` member functions.

8.10 C++11 Numeric Conversions

11 C++11 added functions for converting from numeric values to strings and from strings to numeric values.

Converting Numeric Values to `string` Objects

C++11's `to_string` function (from header `<string>`) returns the string representation of its numeric argument. The function is overloaded for all the fundamental numeric types `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` and `long double`.

Converting `string` Objects to Numeric Values

11 C++11 provides eight functions (from the `<string>` header) for converting `string` objects to numeric values:

Function	Return type	Function	Return type
<i>Functions that convert to integral types</i>		<i>Functions that convert to floating-point types</i>	
<code>stoi</code>	<code>int</code>	<code>stof</code>	<code>float</code>
<code>stol</code>	<code>long</code>	<code>stod</code>	<code>double</code>
<code>stoul</code>	<code>unsigned long</code>	<code>stold</code>	<code>long double</code>
<code>stoll</code>	<code>long long</code>		
<code>stoull</code>	<code>unsigned long long</code>		

Each function attempts to convert the beginning of its `string` argument to a numeric value. If no conversion can be performed, each function throws an `invalid_argument` exception. If the result of the conversion is out of range for the function's return type, each function throws an `out_of_range` exception.

Functions That Convert strings to Integral Types

Consider an example of converting a string to an integral value. Assuming the string:

```
string s("100hello");
```

the following statement converts the beginning of the string to the int value 100 and stores that value in convertedInt:

```
int convertedInt = stoi(s);
```

Each function that converts a string to an integral type receives three parameters—the last two have default arguments. The parameters are:

- A string containing the characters to convert.
- A pointer to a size_t variable. The function uses this pointer to store the index of the first character that was not converted. The default argument is nullptr, in which case the function does not store the index.
- An int from 2 to 36 representing the number's base—the default is base 10.

So, the preceding statement is equivalent to

[Click here to view code image](#)

```
int convertedInt = stoi(s, nullptr, 10);
```

Given a size_t variable named index, the statement:

[Click here to view code image](#)

```
int convertedInt = stoi(s, &index, 2);
```

converts the binary number "100" (base 2) to an int (100 in binary is the int value 4) and stores in index the location of the string's letter "h" (the first character that was not converted).

Functions That Convert strings to Floating-Point Types

The functions that convert strings to floating-point types each receive two parameters:

- A string containing the characters to convert.
- A pointer to a `size_t` variable where the function stores the index of the first character that was not converted. The default argument is `nullptr`, in which case the function does not store the index.

Consider an example of converting a `string` to a floating-point value. Assuming the `string`:

```
string s("123.45hello");
```

the following statement converts the beginning of the `string` to the double value 123.45 and stores that value in `convertedDouble`:

```
double convertedDouble = stod(s);
```

Again, the second argument is `nullptr` by default.

8.11 C++17 `string_view`

¹⁷ C++17 introduced `string_views` (header `<string_view>`), which are read-only views of C-strings or `std::string` objects. Like `std::span`, a `string_view` does not own the data it views. It contains:

- a pointer to the first character in a contiguous sequence of characters and
- a count of the number of characters.

 **PERF CG** `string_views` enable many `std::string`-style operations on C-strings without the overhead of creating and initializing `std::string` objects, which copies the C-string contents. The C++ Core Guidelines state that you should prefer `std::string` if you need to “own character sequences”—for example, to be able to modify `std::string` contents.¹ If you simply need a read-only view of a contiguous sequence of characters, use a `string_view`.²

1. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-string>.

2. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-view>.

Creating a `string_view`

Figure 8.9 demonstrates several `string_view` features. Line 5 includes the

header <string_view>. Line 11 creates the std::string s1 and line 12 copies s1 into s2. Line 12 initializes a string_view with the std::string s1 (line 12), but you can also initialize a string_view with a C-string.

[Click here to view code image](#)

```
1 // fig08_09.cpp
2 // C++17 string_view.
3 #include <iostream>
4 #include <string>
5 #include <string_view>
6 #include "fmt/format.h" // In C++20, this is
7 using namespace std;
8
9 int main() {
10     string s1{"red"};
11     string s2{s1};
12     string_view v1{s1}; // v2 "sees" the contents of s1
13     cout << fmt::format("s1: {}\ns2: {}\n v1: {}",
14 }
```

```
s1: red
s2: red
v1: red
```

Fig. 8.9 C++17 string_view.

string_views “See” Changes to the Characters They View

Because a string_view does not own the sequence of characters it views, it “sees” any changes to the original characters. Line 16 modifies the std::string s1. Then line 17 shows s1’s, s2’s and the string_view v1’s contents.

[Click here to view code image](#)

```
15     // string_views see changes to the characters
16     s1.at(0) = 'R'; // capitalize s1
17     cout << fmt::format("s1: {}\\ns2: {}\\nv1: {}\\n"
18
```

```
< [ ] >
s1: Red
s2: red
v1: Red
```

string_views Are Comparable with std::strings or string_views

Like `std::string`, `string_views` are comparable with the relational and equality operators. You also can intermix `std::strings` and `string_views` as in the equality comparisons in line 21.

[Click here to view code image](#)

```
19     // string_views are comparable with strings or
20     cout << fmt::format("s1 == v1: {}\\ns2 == v1:
21                     s1 == v1, s2 == v1");
22
```

```
< [ ] >
s1 == v1: true
s2 == v1: false
```

string_views Can Remove a Prefix or Suffix

 **PERF** You can easily remove a specified number of characters from the beginning or end of a `string_view`. These are fast operations for a `string_view`—they simply adjust the character count and, in the case of removing from the beginning, move the pointer to the first character in the `string_view`. Lines 24–25 call `string_view` member functions `remove_prefix` and `remove_suffix` to remove one character from the

beginning and one from the end of v1, respectively. Note that s1 remains unmodified.

[Click here to view code image](#)

```
23     // string_view can remove a prefix or suffix
24     v1.remove_prefix(1); // remove one character
25     v1.remove_suffix(1); // remove one character
26     cout << fmt::format("s1: {}\\nv1: {}\\n\\n", s1,
27
```

```
<   >
s1: Red
v1: e
```

[string_views Are Iterable](#)

Line 29 initializes a `string_view` from a C-string. Like `std::strings`, `string_views` are iterable, so you can use them with the range-based `for` statement (as in lines 31–33).

[Click here to view code image](#)

```
28     // string_views are iterable
29     string_view v2{"C-string"};
30     cout << "The characters in v2 are: ";
31     for (const char c : v2) {
32         cout << c << " ";
33     }
34
```

```
The characters in v2 are: C - s t r i n g
```

[string_views Enable Various String Operations on C-Strings](#)

Many `std::string` member functions that do not modify a string also are defined for `string_views`. For example, line 36 calls `size` to determine

the number of characters in the `string_view`, line 37 calls `find` to get the index of '-' in the `string_view`. Line 38 uses the new C++20 `starts_with` function to determine whether the `string_view` starts with 'C'. For a complete list of `string_view` member functions, see

https://en.cppreference.com/w/cpp/string/basic_string_view

[Click here to view code image](#)

```
35     // string_views enable various string operati
36     cout << fmt::format("\n\nv2.size(): {}\\n", v2
37     cout << fmt::format("v2.find('-'): {}\\n", v2.
38     cout << fmt::format("v2.starts_with('C'): {}\\
39 }
```

```
v2.size(): 8
v2.find('-'): 1
v2.starts_with('C'): true
```

8.12 Files and Streams

C++ views each file simply as a sequence of bytes:



Each file ends either with an **end-of-file marker** or at a specific byte number recorded in an operating-system-maintained administrative data structure. When a file is opened, an object is created, and a stream is associated with the object. The objects `cin`, `cout`, `cerr` and `clog` are created for you in the header `<iostream>`. The streams associated with these objects provide communication channels between a program and a particular file or device. The `cin` object (standard input stream object) enables a program to input data from the keyboard or other devices. The `cout` object (standard output

stream object) enables a program to output data to the screen or other devices. The objects `cerr` and `clog` objects (standard error stream objects) enable a program to output error messages to the screen or other devices. Messages written to `cerr` are output immediately. In contrast, messages written to `clog` are stored in a memory object called a buffer. When the buffer is full, its contents are written to the standard error stream.

File-Processing Streams

To perform file processing in C++, headers `<iostream>` and `<fstream>` must be included. Header `<fstream>` includes the following definitions:

- `ifstream` is for file input that reads chars.
- `ofstream` is for file output that writes chars.
- `fstream` combines the capabilities of `ifstream` and `ofstream`.

The `cout` and `cin` capabilities we've discussed so far and the additional I/O features we describe in [Chapter 18](#) also can be applied to file streams.

8.13 Creating a Sequential File

C++ imposes no structure on files. Thus, a concept like that of a record (Section 1.4) does not exist in C++. You must structure files to meet the application's requirements. The following example shows how you can impose a simple record structure on a file.

[Figure 8.10](#) creates a sequential file that might be used in an accounts-receivable system to help keep track of the money owed to a company by its credit clients. For each client, the program obtains the client's account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a record for that client.

[Click here to view code image](#)

```
1 // fig08_10.cpp
2 // Creating a sequential file.
3 #include <cstdlib> // exit function protot
4 #include <fstream> // contains file stream
```

```
5 #include <iostream>
6 #include <string>
7 #include "fmt/format.h" // In C++20, this is
8 using namespace std;
9
10 int main() {
11     // ofstream opens the file
12     if (ofstream output{"clients.txt", ios::out}) {
13         cout << "Enter the account, name, and
14             << "Enter end-of-file to end input
15
16     int account;
17     string name;
18     double balance;
19
20     // read account, name and balance from
21     while (cin >> account >> name >> balance)
22         output << fmt::format("{} {} {}\n");
23     }
24 }
25 else {
26     cerr << "File could not be opened\n";
27     exit(EXIT_FAILURE);
28 }
29 }
```

< >

Enter the account, name, and balance.
Enter end-of-file to end input.

? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

Fig. 8.10 Creating a sequential file.

Opening a File

Figure 8.10 writes data to a file, so we open the file for output by creating an `ofstream` object (line 12). Two arguments are used to initialize the `ofstream`—the **filename** and the **file-open mode**. For an `ofstream` object, the file-open mode can be

- `ios::out` (the default) to output data to a file or
- `ios::app` to append data to the end of a file (without modifying any data already in the file).

11 17 Line 12 creates the `ofstream` object `output` associated with the file `clients.txt` and opens it for output. We did not specify a path to the file (that is, its location), so it's placed in the same directory as the program. Before C++11, the filename was specified as a pointer-based string. C++11 added specifying the filename as a `string` object. C++17 introduced the **<filesystem> header** with features for manipulating files and folders in C++. As of C++17, you also may specify the file to open as a `filesystem::path` object.

Since `ios::out` is the default, the second argument in line 12 is not required, so we could have used:

```
ofstream output{"clients.txt"}
```

Existing files opened with mode `ios::out` are **truncated**—all data in the file is discarded. If the file does not yet exist, the `ofstream` object creates the file. The following table lists the file-open modes—these modes can also be combined:

Mode	Description
<code>ios::app</code>	Append all output to the end of the file.
<code>ios::ate</code>	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
<code>ios::in</code>	Open a file for input.
<code>ios::out</code>	Open a file for output.
<code>ios::trunc</code>	Discard the file's contents (this also is the default action for <code>ios::out</code>).
<code>ios::binary</code>	Open a file for binary, i.e., nontext, input or output.

Opening a File via the `open` Member Function

You can create an `ofstream` object without opening a specific file. In this case, a file can be attached to the object later. For example, the statement

```
ofstream output;
```

creates an `ofstream` object that's not yet associated with a file. The `ofstream` member function `open` opens a file and attaches it to an existing `ofstream` object as follows:

```
output.open("clients.txt", ios::out);
```

Again, `ios::out` is the default value for the second argument. Use caution when opening an existing file for output (`ios::out`), especially when you want to preserve the file's contents, which will be discarded without warning.

Testing Whether a File Was Opened Successfully

11 After creating an `ofstream` object and attempting to open it, the `if` statement uses the file object `output` as a condition (line 12) to determine whether the `open` operation succeeded. For a file object, there is an overloaded operator `bool` (added in C++11) that implicitly evaluates the file object to `true` if the file opened successfully. Some possible reasons opening a file might fail are:

- attempting to open a nonexistent file for reading

- attempting to open a file for reading or writing in a directory that you don't have permission to access, and
-  **Security** opening a file for writing when no secondary storage space is available.

If the condition indicates an unsuccessful attempt to open the file, line 26 outputs an error message, and line 27 invokes function `exit` to terminate the program. The argument to `exit` is returned to the environment from which the program was invoked. Passing `EXIT_SUCCESS` (defined in `<cstdlib>`) to `exit` indicates that the program terminated normally; passing any other value (in this case, `EXIT_FAILURE`) indicates that the program terminated due to an error.

Processing Data

If line 12 opens the file successfully, the program begins processing data. Lines 13–14 prompt the user to enter either the various fields for each record or the end-of-file indicator when data entry is complete. To enter end-of-file key on Linux or macOS type `<Ctrl-d>` (on a line by itself). On Microsoft Windows, type `<Ctrl-z>` then press *Enter*.

The `while` statement's condition (line 21) implicitly invokes the operator `bool` member function on `cin`. The condition remains true as long as each input operation with `cin` is successful. Entering the end-of-file indicator causes the operator `bool` member function to return `false`. You also can call member function `eof` on the input object to determine whether the end-of-file indicator has been entered.

Line 21 extracts each set of data into the variables `account`, `name` and `balance`, and determines whether end-of-file has been entered. When end-of-file is encountered (that is, when the user enters the end-of-file key combination) or an input operation fails, the operator `bool` returns `false`, and the `while` statement terminates.

Line 28 writes a set of data to the file `clients.txt`, using the stream insertion operator `<<` and the output object associated with the file at the beginning of the program. The data may be retrieved by a program designed to read the file (see [Section 8.14](#)). The file created in [Fig. 8.10](#) is simply a text file—it can be viewed by any text editor.

Closing a File

Once the user enters the end-of-file indicator, the `while` loop terminates. At this point, the `output` object goes out of scope, which automatically closes the file. You should always close a file as soon as it's no longer needed in a program. You also can close a file object explicitly, using member function `close`, as in:

```
output.close();
```

Sample Execution

In the sample execution of Fig. 8.10, the user enters information for five accounts, then signals that data entry is complete by entering end-of-file (^Z is displayed for Microsoft Windows). This dialog window does not show how the data records appear in the file. The next section shows how to create a program that reads this file and prints its contents.

8.14 Reading Data from a Sequential File

The previous section demonstrated how to create a sequential-access file. Figure 8.11 reads data sequentially from a file and displays the records from the `clients.txt` file that we wrote in Fig. 8.10. Creating an `ifstream` object opens a file for input. An `ifstream` is initialized with a filename and file-open mode. Line 12 creates an `ifstream` object called `input` that opens the `clients.txt` file for reading. If a file's contents should not be modified, use `ios::in` to open it only for input to prevent unintentional modification of the file's contents.

[Click here to view code image](#)

```
1 // fig08_11.cpp
2 // Reading and printing a sequential file.
3 #include <cstdlib>
4 #include <fstream> // file stream
5 #include <iostream>
6 #include <string>
7 #include "fmt/format.h" // In C++20, this
```

```

8     using namespace std;
9
10    int main() {
11        // ifstream opens the file
12        if (ifstream input{"clients.txt", ios::in})
13            cout << fmt::format("{:<10}{:<13}{}\n"
14                           "Account", "Name", "Balance");
15
16        int account;
17        string name;
18        double balance;
19
20        // display each record in file
21        while (input >> account >> name >> balance)
22            cout << fmt::format("{:<10}{:<13}{"
23                           "account, name, balance);
24        }
25    }
26    else {
27        cerr << "File could not be opened\n";
28        exit(EXIT_FAILURE);
29    }
30}

```



Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

Fig. 8.11 Reading and printing a sequential file.

Opening a File for Input

Objects of class `ifstream` are opened for input by default, so

```
ifstream input("clients.txt");
```

opens `clients.txt` for input. An `ifstream` object can be created without opening a file—you can attach one to it later. Before attempting to retrieve data from the file, line 12 uses the `input` object as a condition to determine whether the file was opened successfully. The overloaded operator `bool` converts the `ifstream` object to a `true` or `false` value.

Reading from the File

Line 21 reads a set of data (i.e., a record) from the file. After line 21 executes the first time, `account` has the value 100, `name` has the value "Jones" and `balance` has the value 24.98. Each time line 21 executes, it reads another record into the variables `account`, `name` and `balance`. Lines 22–23 display the records. When the end of the file is reached, the implicit call to operator `bool` in the `while` condition returns `false`, the `ifstream` object goes out of scope (which automatically closes the file), and the program terminates.

File-Position Pointers

Programs often read sequentially from the beginning of a file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning) during the execution of a program. `istream` and `ostream` provide member functions `seekg` ("seek get") and `seekp` ("seek put") to reposition the **file-position pointer**. This represents the byte number of the next byte in the file to be read or written. Each `istream` object has a **get pointer**, which indicates the byte number in the file from which the next input is to occur. Each `ostream` object has a **put pointer**, which indicates the byte number in the file at which the next output should be placed. The statement

```
input.seekg(0);
```

repositions the file-position pointer to the beginning of the file (location 0) attached to `input`. The argument to `seekg` is an integer. If the end-of-file indicator has been set, you'd also need to execute

```
input.clear();
```

to re-enable reading from the stream.

An optional second argument indicates the **seek direction**:

- **ios::beg** (the default) for positioning relative to the beginning of a stream,
- **ios::cur** for positioning relative to the current position in a stream or
- **ios::end** for positioning backward relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as the number of bytes from the file's starting location. This is also referred to as the **offset** from the beginning of the file. Some examples of positioning the *get* file-position pointer are

[Click here to view code image](#)

```
// position to the nth byte of fileObject (assume:  
fileObject.seekg(n);  
  
< >  
  
// position n bytes in fileObject  
fileObject.seekg(n, ios::cur);  
  
// position n bytes back from end of fileObject  
fileObject.seekg(n, ios::end);  
  
// position at end of fileObject  
fileObject.seekg(0, ios::end);
```

The same operations can be performed using `ostream` member function `seekp`. Member functions `tellg` and `tellp` are provided to return the current locations of the *get* and *put* pointers, respectively. The following statement assigns the *get* file-position pointer value to variable `location` of type `long`:

```
location = fileObject.tellg();
```

8.15 C++14 Reading and Writing Quoted Text

14 Many text files contain quoted text, such as "C++20 for Programmers". For example, in files representing HTML5 web pages, attribute values are enclosed in quotes. If you're building a web browser to

display the contents of such a web page, you must be able to read those quoted strings and remove the quotes.

Suppose you need to read from a text file, as you did in Fig. 8.11, but with each account's data formatted as follows:

```
100 "Janie Jones" 24.98
```

Recall that the stream extraction operator `>>` treats white space as a delimiter. So, if we read the preceding data using the expression in line 30 of Fig. 8.11:

```
input >> account >> name >> balance
```

the first stream extraction reads 100 into the int variable `account`, and the second reads only "Janie into the string variable `name`. The opening double quote would be part of the string in `name`. The third stream extraction fails while attempting to read a value for the double variable `balance` because the next token (i.e., piece of data) in the input stream—Jones"—is not a double.

Reading Quoted Text

14 C++14 added the stream manipulator `quoted` (header `<iomanip>`) for reading quoted text from a stream. It includes any white space characters in the quoted text and discards the double-quote delimiters. For example, if we read the preceding data, the expression:

[Click here to view code image](#)

```
input >> account >> quoted(name) >> balance
```

reads 100 into `account`, reads Janie Jones as one string into `name`, and reads 24.98 into `balance`. If the quoted data contains \" escape sequences, each is read and stored in the string as the escape sequence \"—not as ".

Writing Quoted Text

Similarly, you can write quoted text to a stream. For example, if `name` contains Janie Jones, the statement:

```
outputStream << quoted(name);
```

writes to the text-based `OutputStream`:

```
"Janie Jones"
```

8.16 Updating Sequential Files

Data that is formatted and written to a sequential file, as shown in [Section 8.13](#), cannot be modified in place without the risk of destroying other data in the file. For example, if the name “White” needs to be changed to “Worthington,” the old name cannot be overwritten without corrupting the file. The record for White was written to the file as

```
300 White 0.00
```

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

```
300 Worthington 0.00
```

The new record contains six more characters than the original. Any characters after “h” in “Worthington” would overwrite 0.00 and the beginning of the next sequential record in the file. The problem with the formatted input/output model using the stream insertion operator `<<` and the stream extraction operator `>>` is that fields—and hence records—can vary in size. For example, values 7, 14, -117, 2074, and 27383 are all `ints`, which store the same number of “raw data” bytes internally (typically four bytes on 32-bit machines and eight bytes on 64-bit machines). However, these integers become different-sized fields, depending on their actual values, when output as formatted text (character sequences). Therefore, the formatted input/output model usually is not used to update records in place.

Such updating can be done with sequential files, but awkwardly. For example, to make the preceding name change in a sequential file, we could:

- copy the records before 300 White 0.00 to a new file,
- write the updated record to the new file, then
- write the records after 300 White 0.00 to the new file.

Then we could delete the old file and rename the new one. This requires processing every record in the file to update one record. If many records are

being updated in one pass of the file, though, this technique can be acceptable.

8.17 String Stream Processing

In addition to standard stream I/O and file stream I/O, C++ stream I/O includes capabilities for inputting from, and outputting to, strings in memory. These capabilities often are referred to as **in-memory I/O** or **string stream processing**. You can read from a string with ***istringstream*** and write to a string with ***ostringstream***.

Class templates *istringstream* and *ostringstream* provide the same functionality as classes *istream* and *ostream* plus other member functions specific to in-memory formatting. Programs that use in-memory formatting must include the `<sstream>` and `<iostream>` headers. An *ostringstream* object uses a *string* object to store the output data. Its ***str*** member function returns a copy of that *string*.

One application of string stream processing is data validation. A program can read an entire line at a time from the input stream into a *string*. Next, a validation routine can scrutinize the *string*'s contents and correct (or repair) the data, if necessary. Then the program can input from the *string*, knowing that the input data is in the proper format.

11 To assist with data validation, C++11 added powerful pattern-matching regular-expression capabilities. For instance, in a program requiring a U.S. format telephone number (e.g., (800) 555-1212), you can use a regular expression to confirm that a *string* matches that format. Many websites provide regular expressions for validating e-mail addresses, URLs, phone numbers, addresses and other popular kinds of data. We introduce regular expressions and provide several examples in [Section 8.20](#).

Demonstrating *ostringstream*

Figure 8.12 creates an *ostringstream* object, then uses the stream insertion operator to output a series of *strings* and numerical values to the object.

[Click here to view code image](#)

```
1 // fig08_12.cpp
2 // Using an ostringstream object.
3 #include <iostream>
4 #include <sstream> // header for string st
5 #include <string>
6 using namespace std;
7
8 int main() {
9     ostringstream output; // create ostrings
10
11    const string string1{"Output of several '};
12    const string string2{"to an ostringstream"};
13    const string string3{"\n      double: "};
14    const string string4{"\n      int: "};
15
16    constexpr double s{123.4567};
17    constexpr int i{22};
18
19    // output strings, double and int to ostream
20    output << string1 << string2 << string3;
21
22    // call str to obtain string contents of
23    cout << "output contains:\n" << output.str();
24
25    // add additional characters and call str
26    output << "\nmore characters added";
27    cout << "\n\nafter additional stream inser
28    << "" << output.str() << endl;
29 }
```



```
outputString contains:
Output of several data types to an ostringstream
double: 123.457
int: 22
```

```
after additional stream insertions, outputString  
Output of several data types to an ostringstream  
double: 123.457  
int: 22  
more characters added
```



Fig. 8.12 Using an `ostringstream` object.

Line 20 outputs string `string1`, `string string2`, `string string3`, double `d`, string `string4` and int `i`—all to output in memory. Line 23 displays `output.str()`, which returns the string created by `output` in line 20. Line 25 appends more data to the string in memory by simply issuing another stream insertion operation to `output`, then lines 27–28 display the updated contents.

Demonstrating `istringstream`

An `istringstream` object inputs data from a string in memory. Data is stored in an `istringstream` object as characters. Input from the `istringstream` object works identically to input from any file. The end of the string is interpreted by the `istringstream` object as end-of-file.

Figure 8.13 demonstrates input from an `istringstream` object. Lines 9–10 create string `inputString` containing the data and `istringstream` object `input` constructed to read from `inputString`, which consists of two strings ("Amanda" and "test"), an int (123), a double (4.7) and a char ('A'). These are read into variables `s1`, `s2`, `i`, `d` and `c` in line 18, then displayed in lines 20–21. Next, the program attempts to read from `input` again in line 24, but the operation fails because there is no more data in `inputString`. So the `input` evaluates to `false`, and the `else` part of the `if...else` statement executes.

[Click here to view code image](#)

```
1 // fig08_13.cpp
2 // Demonstrating input from an istringstream
3 #include <iostream>
```

```
Items extracted from the istringstream object:  
Amanda  
test  
123  
4.7  
A  
  
input is empty
```

Fig. 8.13 Demonstrating input from an `istringstream` object.

8.18 Raw String Literals

Recall that backslash characters in strings introduce escape sequences—like `\n` for newline and `\t` for tab. If you wish to include a backslash in a string, you must use two backslash characters `\\"`, making some strings difficult to read. For example, Microsoft Windows uses backslashes to separate folder names when specifying a file's location. To represent a file's location on Windows, you might write:

[Click here to view code image](#)

```
string windowsPath{ "C:\\MyFolder\\MySubFolder\\My":  
    < >
```

For such cases, **raw string literals** (introduced in C++11) that have the format

`R"(rawCharacters)"`

are more convenient. The parentheses are required around the *rawCharacters* that compose the raw string literal. The compiler automatically inserts backslashes as necessary in a raw string literal to properly escape special characters like double quotes ("), backslashes (\), etc. Using a raw string literal, we can write the preceding string as:

[Click here to view code image](#)

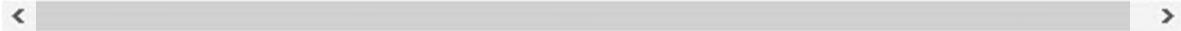
```
string windowsPath{R"(C:\\MyFolder\\MySubFolder\\MyF":  
    < >
```

Raw strings can make your code more readable, particularly when using the regular expressions that we discuss in [Section 8.20](#). Regular expressions often contain many backslash characters. We'll also use raw strings in [Section 8.19.2](#).

The preceding raw string literal can include optional delimiters up to 16 characters long before the left parenthesis, (, and after the right parenthesis,), as in

[Click here to view code image](#)

```
R"MYDELIMITER (J.*\d[0-35-9]-\d\d-\d\d) MYDELIMITER"
```



The optional delimiters must be identical if provided.

Raw string literals may be used in any context that requires a string literal. They may also include line breaks, in which case the compiler inserts \n escape sequences. For example, the raw string literal

```
R" (multiple lines  
of text)"
```

is treated as the string literal

```
"multiple lines\nof text"
```

8.19 Objects Natural Case Study: Reading and Analyzing a CSV File Containing Titanic Disaster Data

The **CSV (comma-separated values)** file format, which uses the **.csv file extension**, is particularly popular, especially for datasets used in big data, data analytics and data science and in artificial intelligence applications like machine learning and deep learning. Here, we'll demonstrate reading from a CSV file.

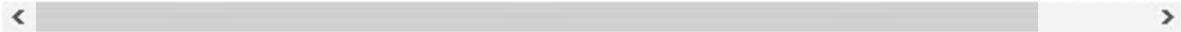
Datasets

There's an enormous variety of free datasets available online. The OpenML machine learning resource site

<https://openml.org>

contains over 21,000 free datasets in comma-separated values (CSV) format. Another great source of datasets is:

<https://github.com/awesomedata/awesome-public-datasets>



[account.csv](#)

For our first example, we've included a simple dataset in `accounts.csv` in the `ch08` folder. This file contains the account information shown in Fig. 8.11's output, but in the format:

```
account,name,balance
100,Jones,24.98
200,Doe,345.67
300,White,0.0
400,Stone,-42.16
500,Rich,224.62
```

The first row of a CSV file typically contains column names. Each subsequent row contains one data record representing the values for those columns. In this dataset, we have three columns representing an account, name and balance.

8.19.1 Using `rapidcsv` to Read the Contents of a CSV File

The `rapidcsv`³ header-only library

3. Copyright ©2017, Kristofer Berggren. All rights reserved.

<https://github.com/d99kris/rapidcsv>

provides class `rapidcsv::Document` that you can use to read and manipulate CSV files. Many other libraries have built-in CSV support. For your convenience, we provided `rapidcsv` in the example's folder's `libraries/rapidcsv` subfolder. As in earlier examples that use open-source libraries, you'll need to point the compiler at the `rapidcsv` subfolder's `src` folder so you can include "`rapidcsv.h`" (Fig. 8.14, line 7).

[Click here to view code image](#)

```
1 // fig08_14.cpp
2 // Reading from a CSV file.
3 #include <iostream>
4 #include <string>
5 #include <vector>
```

```

6   #include "fmt/format.h" // In C++20, this
7   #include "rapidcsv.h"
8   using namespace std;
9
10  int main() {
11      rapidcsv::Document document{"accounts.csv"};
12      vector<int> accounts{document.GetColumn<
13          vector<string> names{document.GetColumn<
14          vector<double> balances{document.GetColu
15
16      cout << fmt::format("{:<10}{:<13}{}\n",
17
18      for (size_t i{0}; i < accounts.size(); +
19          cout << fmt::format("{:<10}{:<13}{:>7
20                      accounts.at(i), names.at(i
21      }
22  }

```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

Fig. 8.14 Reading from a CSV file.

Line 11 creates and initializes a `rapidcsv::Document` object named `document`. This statement loads the specified file ("accounts.csv"). Class `Document`'s member functions enable you to work with the CSV data by row, by column or by individual value in a specific row and column. In this example, lines 12–14 get the data using the class's `GetColumn` template member function. This function returns the specified column's data as a `std::vector` containing elements of the type you specify in angle brackets. Line 12's call

```
document.GetColumn<int>("account")
```

returns a `vector<int>` containing the account numbers for every record. Similarly, the calls in line 13–14 return a `vector<string>` and a `vector<double>` containing all the records' names and balances, respectively. Lines 16–21 format and display the file's contents to confirm that they were read properly.

Caution: Commas in CSV Data Fields

Be careful when working with strings containing embedded commas, such as the name "Jones, Sue". If this name were accidentally stored as the two strings "Jones" and "Sue", that CSV record would have *four* fields, not *three*. Programs that read CSV files typically expect every record to have the same number of fields; otherwise, problems occur.

Caution: Missing Commas and Extra Commas in CSV Files

Be careful when preparing and processing CSV files. For example, suppose your file is composed of records, each with *four* comma-separated `int` values, such as:

```
100, 85, 77, 9
```

If you accidentally omit one of these commas, as in:

```
100, 8577, 9
```

then the record has only *three* fields, one with the invalid value 8577.

If you put two adjacent commas where only one is expected, as in:

```
100, 85,, 77, 9
```

then you have *five* fields rather than *four*, and one of the fields erroneously would be *empty*. Each of these comma-related errors could confuse programs trying to process the record.

8.19.2 Reading and Analyzing the Titanic Disaster Dataset

DS One of the most commonly used datasets for data analytics and data science beginners is the **Titanic disaster dataset**⁴. It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank

on its maiden voyage April 14–15, 1912. We'll load the dataset in Fig. 8.15, view some of its data and perform some basic data analytics.

4. “Titanic” dataset on OpenML.org (<https://www.openml.org/d/40945>). Author: Frank E. Harrell, Jr. and Thomas Cason. Source: Vanderbilt Biostatistics (<http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.html>). The OpenML license terms (<https://www.openml.org/cite>) say, “You are free to use OpenML and all empirical data and metadata under the CC-BY license (<http://creativecommons.org/licenses/by/4.0/>), requesting appropriate credit if you do.”

[Click here to view code image](#)

```
1 // fig08_14.cpp
2 // Reading the Titanic dataset from a CSV
3 #include <algorithm>
4 #include <cmath>
5 #include <iostream>
6 #include <numeric>
7 #include <ranges>
8 #include <string>
9 #include <vector>
10 #include "fmt/format.h" // In C++20, this is
11 #include "rapidcsv.h"
12 using namespace std;
13
14 int main() {
15     // load Titanic dataset; treat missing values as NaN
16     rapidcsv::Document titanic("titanic.csv");
17     rapidcsv::LabelParams{}, rapidcsv::StringType::kUTF8,
18     rapidcsv::ConverterParams{true});
19 }
```



Fig. 8.15 Reading the Titanic dataset from a CSV file, then analyzing it.

To download the dataset in CSV format go to

<https://www.openml.org/d/40945>

and click the CSV download button in the page's upper-right corner. This downloads the file `phpMYEkM1.csv`, which we renamed as `titanic.csv`. We assume that you'll download the file, rename it as `titanic.csv` and place it in the chapter's `ch08` examples folder.

20 Figure 8.14 uses some C++20 ranges library features we introduced in Section 6.14. At the time of this writing, only GNU C++ 10.1 supports the ranges library.

Getting to Know the Data

Much of data analytics and data science is devoted to getting to know your data. One way is simply to look at the raw data. If you open the `titanic.csv` file in a text editor or spreadsheet application, you'll see that the dataset contains 1309 rows, each containing 14 columns—often called **features** in data analytics. We'll use only four columns here:

- `survived`: 1 or 0 for yes or no, respectively.
- `sex`: "female" or "male".
- `age`: The passenger's age. Most values in this column are integers, but some children under 1 year of age have floating-point `age` values, so we'll process this column as `double` values.
- `pclass`: 1, 2 or 3 for first class, second class or third class, respectively.

To learn more about the dataset's origins and its other columns, visit:

<http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.csv>

Missing Data

Bad data values and missing values can significantly impact data analysis. The Titanic dataset is missing ages for 263 passengers. These are represented as ? in the CSV file. In this example, when we produce descriptive statistics for the passengers' ages, we'll filter out and ignore the missing values. Some data scientists advise against any attempts to insert "reasonable values." Instead, they advocate clearly marking missing data and leaving it up to a data analytics package to handle the issue. Others offer strong cautions.⁵

⁵ This footnote was abstracted from a comment sent to us July 20, 2018 by one of our Python textbook's academic reviewers, Dr. Alison Sanchez of the University of San Diego School of Business. She commented: "Be cautious when mentioning 'substituting reasonable values' for missing or bad values.' A stern warning: 'Substituting' values that increase statistical significance or give more 'reasonable' or 'better' results is not permitted. 'Substituting' data should not turn into 'fudging' data. The first rule students should learn is not to eliminate or change values that contradict their hypotheses. 'Substituting reasonable values' does not mean students should feel free to change values to get the results they want."

Loading the Dataset

Lines 16–18 create and initialize a `rapidcsv::Document` object named `titanic` that loads "titanic.csv". The second and third arguments in line 17 are two of the default arguments used to initialize a `Document` object when you create it only by specifying the CSV file name. Recall from our discussion of default function arguments that when an argument is specified explicitly for a parameter with a default argument, all prior arguments in the argument list also must be specified explicitly. We provided the second and third arguments, so we can specify the fourth argument.

The `rapidcsv::LabelParams{}` argument specifies by default that the CSV file's first row contains the column names. The `rapidcsv::SeparatorParams{}` argument specifies by default that each record's fields are separated by commas. The fourth argument:

```
rapidcsv::ConverterParams{true}
```

enables RapidCSV to convert missing and bad data values in integer columns to 0 and floating-point columns to NaN (not a number). This enables us to load all the data in the age column into a `vector<double>`, including the missing values represented by ?.

Removing the Quotes from the Columns Containing Strings

Lines 21–24 use the `rapidcsv::Document`'s `GetColumn` member function to get each column by name. In the dataset, the column names are enclosed in double-quote characters, so they are required to load the columns. For example, in the raw string `R("survived")` represents the column name "survived" with the enclosing quotes.

[Click here to view code image](#)

```
20    // get data by column; ignoring column 0 in t
```

```
21     auto survived{titanic.GetColumn<int>(R"("survived"))}
22     auto sex{titanic.GetColumn<string>(R"("sex"))}
23     auto age{titanic.GetColumn<double>(R"("age"))}
24     auto pclass{titanic.GetColumn<int>(R"("pclass"))}
25
```

Removing the Quotes from the Column Containing Strings

String values in the Titanic dataset are enclosed in double-quotes—e.g., "female" and "male". The `rapidcsv::Document` object includes the quotes in each such `std::string` it returns. Lines 27–29 remove these quotes from each value in the vector named `sex`. For the current `item`, line 28 calls `std::string` member function `erase` to remove the character at index 0 (a quote) in the string. The first argument is the index (0) at which to begin erasing characters. The second is the number of characters to erase (1). `erase` returns a reference to the modified `std::string`, which we use to the `std::string`'s `pop_back` member function. This removes the string's last character (also a quote).

[Click here to view code image](#)

```
26     // lambda to remove the quotes from the strings
27     for (string& item : sex) {
28         item.erase(0, 1).pop_back();
29     }
30
```

Viewing Some Rows in the Titanic Dataset

The 1309 rows each represent one passenger. According to Wikipedia, there were approximately 1317 passengers and 815 of them died.⁶ For large datasets, it's not possible to display all the data at once. A common practice when getting to know your data is to display a few rows from the beginning and end of the dataset, so you can get a sense of the data. The code in lines 32–37 displays the first five elements of each column's data:

6. https://en.wikipedia.org/wiki/Passengers_of_the_RMS_Titanic.

[Click here to view code image](#)

```
31     // display first 5 rows
32     cout << fmt::format("First five rows:\n{:<10}"
33                     "survived", "sex", "age", "class")
34     for (size_t i{0}; i < 5; ++i) {
35         cout << fmt::format("{:<10}{:<8}{:<6.1f}{"
36                     survived.at(i), sex.at(i), age.at(i), p
37     }
38
```

```
< [ ] >
First five rows:
survived  sex      age    class
1          female   29.0   1
1          male     0.9    1
0          female   2.0    1
0          male     30.0   1
0          female   25.0   1
```

The code in lines 40–46 displays the last five elements of each column’s data. To determine the control variable’s starting value, line 42 calls the `rapidcsv::Document`’s `GetRowCount` member function. Then line 43 initializes the control variable to five less than the row count. Note the value in the `age` column for one of row is `nan`, indicating a missing value in the dataset.

[Click here to view code image](#)

```
39     // display last 5 rows
40     cout << fmt::format("\nLast five rows:\n{:<10}"
41                     "survived", "sex", "age", "class");
42     auto count{titanic.GetRowCount()};
43     for (size_t i{count - 5}; i < count; ++i) {
44         cout << fmt::format("{:<10}{:<8}{:<6.1f}{"
45                     survived.at(i), sex.at(i), age.at(i), p
```

```
46      }
```

```
47
```

```
Last five rows:  
survived  sex    age   class  
0         female  14.5  3  
0         female  nan   3  
0         male   26.5  3  
0         male   27.0  3  
0         male   29.0  3
```

Basic Descriptive Statistics

DS 20 As part of getting to know a dataset, data scientists often use statistics to describe and summarize data. Let's calculate several **descriptive statistics** for the age column, including the number of passengers for which we have age values, and the average, minimum, maximum and median age values. Before performing these calculations, we must remove nan values—calculations performed with nan result in nan. Lines 49–50 use the C++20 ranges filtering techniques from [Section 6.14](#) to keep only the values in the age vector that are *not* nan. Function `isnan` (header `<cmath>`) returns true if the value is nan. Next, line 51 creates a `vector<double>` named `cleanAge`. The vector initializes its elements by iterating through the filtered results from `begin(removeNaN)` to `end(removeNaN)`.

[Click here to view code image](#)

```
48 // use C++20 ranges to eliminate missing values  
49 auto removeNaN =  
50     age | views::filter([](const auto& x) { ret  
51     vector<double> cleanAge{begin(removeNaN), end  
52 }
```

Basic Descriptive Statistics for the Cleaned Age Column

Now, we can calculate the descriptive statistics. Line 54 sorts `cleanAge`, which will help us determine the minimum, maximum and median values. To count the number of people for which we have valid ages, we simply get `cleanAge`'s size (line 55).

[Click here to view code image](#)

```
53     // descriptive statistics for cleaned ages co
54     sort(begin(cleanAge), end(cleanAge));
55     size_t size{cleanAge.size()};
56     double median{};
57
58     if (size % 2 == 0) { // find median value for
59         median = (cleanAge.at(size / 2 - 1) + clea
60     }
61     else { // find median value for odd number of
62         median = cleanAge.at(size / 2);
63     }
64
65     cout << "\nDescriptive statistics for the age
66             << fmt::format("Passengers with age data:
67             << fmt::format("Average age: {:.2f}\n",
68                     accumulate(begin(cleanAge), end(clea
69             << fmt::format("Minimum age: {:.2f}\n", cl
70             << fmt::format("Maximum age: {:.2f}\n", cl
71             << fmt::format("Median age: {:.2f}\n", med
72
```

```
< >
Descriptive statistics for the age column:
Passengers with age data: 1046
Average age: 29.88
Minimum age: 0.17
Maximum age: 80.00
Median age: 28.00
```

Lines 56–63 determine the median. If `cleanAge`'s size is even, the

median is the average of the two middle elements (line 59); otherwise, it's the middle element (line 62). Lines 65–71 display the descriptive statistics. Line 68 calculates the average age by using the accumulate algorithm to total the ages, then dividing the result by size. The vector is sorted, so lines 69–70 determine the minimum and maximum values by calling the vector's **front** and **back** member functions, respectively. The average and median are **measures of central tendency**. Each is a way to produce a single value that represents a “central” value in a set of values, i.e., a value which is, in some sense, typical of the others.

For the 1046 people with valid ages, the average age was 29.88 years old. The youngest passenger (i.e., the minimum) was just over two months old ($0.17 * 12$ is 2.04), and the oldest (i.e., the maximum) was 80. The median age was 28.

Determining Passenger Counts By Class

Let's calculate each class's number of passengers. Lines 74–79 define a lambda that

- filters the pclass column, keeping only the elements for the specified classNumber (line 76),
- maps each to the value 1 (line 77), then
- uses accumulate to total the mapped results, which gives us the count for the specified classNumber.

The lambda introducer [classNumber] in line 76 indicates that the lambda will use the variable classNumber (line 74 in the parameter list) in line 76's lambda body. Known as **capturing** the variable, this is required to use a variable that's not defined in the lambda. Lines 81–83 define constants for the three passenger classes. Lines 84–86 call countClass for each passenger class, and lines 88–90 display the counts.

[Click here to view code image](#)

```
73    // passenger counts by class
74    auto countClass = [](const auto& column, const
75        auto filterByClass{column
76            | views::filter([classNumber](auto x) {
```

```
77         | views::transform([](auto x) {return 1
78             return accumulate(begin(filterByClass), en
79         );
80
81     constexpr int firstClass{1};
82     constexpr int secondClass{2};
83     constexpr int thirdClass{3};
84     const int firstCount{countClass(pclass, first
85     const int secondCount{countClass(pclass, seco
86     const int thirdCount{countClass(pclass, third
87
88     cout << "\nPassenger counts by class:\n"
89     << fmt::format("1st: {}\\n2nd: {}\\n3rd: {}\\n",
90                     firstCount, secondCount, thirdCount)
91
```

< >

```
Passenger counts by class:
1st: 323
2nd: 554
3rd: 2127
```

Basic Descriptive Statistics for the Cleaned Age Column

Let's say you want to determine some statistics about people who survived. Line 93 filters the survived column, keeping only the survivors. Recall that this column contains 1 or 0 to represent survived or died, respectively. These also are values that C++ can treat as true (1) or false (0), so the lambda in line 93 simply returns the column value. If that value is 1, the filter operation keeps that value in the results. Line 94 uses the accumulate function to calculate the survivorCount by iterating over the results and totaling the 1s. To find out how many people died, we simply subtract survivorCount from the survived vector's size. Line 99 calculates the percentage of people who survived.

[Click here to view code image](#)

```
92     // percentage of people who survived
93     auto survivors = survived | views::filter([](
94         int survivorCount{accumulate(begin(survivors)
95
96         cout << fmt::format("\nSurvived count: {}\\nDi
97                 survivorCount, survived.size() - s
98         cout << fmt::format("Percent who survived: {::
99                 100.0 * survivorCount / survived.s
100
```

< >

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
```

Counting By Sex and By Passenger Class the Numbers of People Who Survived

14 Lines 102–122 iterate through the survived column, using its 1 or 0 value as a condition (line 109). For each survivor, we increment counters for the survivor's sex (survivingWomen and survivingMen in line 110) and pclass (surviving1st, surviving2nd and surviving3rd in lines 112–120). We'll use these counts to calculate percentages. In line 110, the literal value "female"s with the trailing s after the closing quote is a C++14 **string object literal** that treats "female" as a std::string rather than a C-string. So the == comparison is between two std::string objects.

[Click here to view code image](#)

```
101    // count who survived by male/female, 1st/2nd
102    int survivingMen{0};
103    int survivingWomen{0};
104    int surviving1st{0};
105    int surviving2nd{0};
106    int surviving3rd{0};
```

```
107
108     for (size_t i{0}; i < survived.size(); ++i)
109         if (survived.at(i)) {
110             sex.at(i) == "female"s ? ++survivingWomen
111
112             if (firstClass == pclass.at(i)) {
113                 ++surviving1st;
114             }
115             else if (secondClass == pclass.at(i))
116                 ++surviving2nd;
117             }
118             else { // third class
119                 ++surviving3rd;
120             }
121         }
122     }
123 }
```



Calculating Percentages of People Who Survived

Lines 125–135 calculate and display the percentages of:

- women who survived,
- men who survived,
- first-class passengers who survived,
- second-class passengers who survived, and
- third-class passengers who survived.

Of the survivors, about two-thirds were women, and first-class passengers survived at a higher rate than the other passenger classes.

[Click here to view code image](#)

```
124     // percentages who survived by male/female,
125     cout << fmt::format("Female survivor percent"
126                           100.0 * survivingWomen / survivor
127                           << fmt::format("Male survivor percentag
```

```
128          100.0 * survivingMen / survivorCo
129          << fmt::format("1st class survivor perc
130              100.0 * surviving1st / survivorCo
131          << fmt::format("2nd class survivor perc
132              100.0 * surviving2nd / survivorCo
133          << fmt::format("3rd class survivor perc
134              100.0 * surviving3rd / survivorCo
135      }
```

Female survivor percentage: 67.80%

Male survivor percentage: 32.20%

1st class survivor percentage: 40.00%

2nd class survivor percentage: 23.80%

3rd class survivor percentage: 36.20%

8.20 Objects Natural Case Study: Introduction to Regular Expressions

Sometimes you'll need to recognize patterns in text, like phone numbers, e-mail addresses, ZIP Codes, web page addresses, Social Security numbers and more. A **regular expression** string describes a search pattern for matching characters in other strings.

Regular expressions can help you extract data from unstructured text, such as social media posts. They're also important for ensuring that data is in the proper format before you attempt to process it.

Validating Data

Before working with text data, you'll often use regular expressions to validate it. For example, you might check that:

- A U.S. ZIP Code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775).
- A string last name contains only letters, spaces, apostrophes and hyphens.

- An e-mail address contains only the allowed characters in the allowed order.
- A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers that can be used in each group of digits.

You'll rarely need to create your own regular expressions for common items like these. The following free websites

- <https://regex101.com>
- <https://regexr.com/>
- <http://www.regexlib.com>
- <https://www.regular-expressions.info>

and others offer repositories of existing regular expressions that you can copy and use. Many sites like these also allow you to test regular expressions to determine whether they'll meet your needs.

Other Uses of Regular Expressions

Regular expressions also are used to:

- Extract data from text (known as *scraping*)—e.g., locating all URLs in a web page.
- Clean data—e.g., removing data that's not required, removing duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, removing formatting, changing text case, dealing with outliers and more.
- Transform data into other formats—e.g., reformatting tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format. We discussed CSV in [Section 8.19](#).

8.20.1 Matching Complete Strings to Patterns

To use regular expressions, include the header `<regex>`, which provides several classes and functions for recognizing and manipulating regular expressions.

Matching Literal Characters

The `<regex>` function `regex_match` (Fig. 8.16) returns `true` if the entire string in its first argument *matches the pattern in its second argument*. By default, pattern matching is case sensitive—later, you’ll see how to perform case-insensitive matches. Let’s begin by matching literal characters—that is, characters that match themselves. Line 10 creates a `regex` object `r1` for the pattern “02215” containing only literal digits that match themselves in the specified order. Line 13 calls `regex_match` for the strings “02215” and “51220”. Each has the same digits, but only “02215” has them *in the correct order for a match*.

[Click here to view code image](#)

```
1 // fig08_16.cpp
2 // Matching entire strings to regular expressions
3 #include <iostream>
4 #include <regex>
5 #include "fmt/format.h" // In C++20, this is part of the standard library
6 using namespace std;
7
8 int main() {
9     // fully match a pattern of literal characters
10    regex r1("02215");
11    cout << "Matching against: 02215\n"
12        << fmt::format("02215: {}; 51220: {}",
13                      regex_match("02215", r1), regex_match("51220", r1));
14}
```

```
< >
Matching against: 02215
02215: true; 51220: false
```

Fig. 8.16 Matching entire strings to regular expressions.

Metacharacters, Character Classes and Quantifiers

Regular expressions typically contain various special symbols called

metacharacters:

[] { } () \ * + ^ \$? . |

The **\ metacharacter** begins each of the predefined **character classes**, several of which are shown in the following table with the groups of characters they match.

Character class	Matches
\d	Any digit (0–9).
\D	Any character that is not a digit.
\s	Any whitespace character (such as spaces, tabs and newlines).
\S	Any character that is not a whitespace character.
\w	Any word character (also called an alphanumeric character)—that is, any uppercase or lowercase letter, any digit or an underscore
\W	Any character that is not a word character.

To match any metacharacter as its *literal value*, precede it by a backslash. For example, \\$ matches a dollar sign (\$) and \\ matches a backslash (\).

Matching Digits

Let's validate a five-digit ZIP Code. In the regular expression \d{5} (created by the raw string literal in line 16), \d is a character class representing a digit (0–9). A character class is a **regular expression escape sequence** that *matches one character*. To match more than one, follow the character class with a **quantifier**. The quantifier {5} repeats \d five times, as if we had written \d\d\d\d\d, to match five consecutive digits. Line 19's second call to `regex_match` returns `false` because "9876" contains only four consecutive digit characters.

[Click here to view code image](#)

```
15    // fully match five digits
16    regex r2(R"(\d{5})");
```

```
17     cout << R"(Matching against: \d{5})" << "\n"
18     << fmt::format("02215: {}; 9876: {})\n\n"
19             regex_match("02215", r2), regex_ma
20
```

```
< ━━━━━━ >
Matching against: \d{5}
02215: true; 9876: false
```

Custom Character Classes

Characters in square brackets, [], define a **custom character class** that *matches a single character*. For example, [aeiou] matches a lowercase vowel, [A-Z] matches an uppercase letter, [a-z] matches a lowercase letter and [a-zA-Z] matches any lowercase or uppercase letter. Let's use a custom character class to validate a simple first name with no spaces or punctuation.

[Click here to view code image](#)

```
21 // match a word that starts with a capital le
22 regex r3("[A-Z][a-z]*");
23 cout << "Matching against: [A-Z][a-z]*\n"
24     << fmt::format("Wally: {}; eva: {})\n\n",
25             regex_match("Wally", r3), regex_ma
26
```

```
< ━━━━━━ >
Matching against: [A-Z][a-z]*
Wally: true; eva: false
```

A first name might contain many letters. In the regular expression `r3` (line 22), [A-Z] matches one uppercase letter, and [a-z]* matches any number of lowercase letters. The *** quantifier** *matches zero or more occurrences of the subexpression to its left* (in this case, [a-z]). So [A-Z][a-z]* matches "Amanda", "Bo" or even "E".

When a custom character class starts with a **caret (^)**, the class *matches any character that's not specified*. So `[^a-z]` (line 28) matches any character that's not a lowercase letter:

[Click here to view code image](#)

```
27 // match any character that's not a lowercase
28 regex r4("[^a-z]");
29 cout << "Matching against: [^a-z]\n"
30     << fmt::format("A: {}; a: {}\\n\\n",
31                     regex_match("A", r4), regex_
32
```

```
< >
Matching against: [^a-z]
A: true; a: false
```

Metacharacters in a custom character class are treated as literal characters —that is, the characters themselves. So `[*+$]` (line 34) matches a single *, + or \$ character:

[Click here to view code image](#)

```
33 // match metacharacters as literals in a cust
34 regex r5("[*+$]");
35 cout << "Matching against: [*+$]\\n"
36     << fmt::format("*: {}; !: {}\\n\\n",
37                     regex_match("*", r5), regex_
38
```

```
< >
Matching against: [*+$]
*: true; !: false
```

* vs. + Quantifier

If you want to require at least one lowercase letter in a first name, replace the

* quantifier in line 22 with **+**. This *matches at least one occurrence of a subexpression to its left*:

[Click here to view code image](#)

```
39 // matching a capital letter followed by at l
40 regex r6("[A-Z][a-z]+");
41 cout << "Matching against: [A-Z][a-z]*\n"
42     << fmt::format("Wally: {}; E: {}\\n\\n",
43                 regex_match("Wally", r6), regex_ma
44
```

Matching against: [A-Z] [a-z]*
Wally: true; E: false

Both * and + are **greedy**—they *match as many characters as possible*. So the regular expression [A-Z] [a-z]+ matches "Al", "Eva", "Samantha", "Benjamin" and any other words that begin with a capital letter followed at least one lowercase letter.

Other Quantifiers

The **? quantifier** matches zero or one occurrence of the subexpression to its left. In the regular expression `label1?ed` (line 46), the subexpression is the literal character "l". So in line 49's `regex_match` calls, the regular expression matches `labelled` (the U.K. English spelling) and `labeled` (the U.S. English spelling), but in line 50's `regex_match` calls, the regular expression does not match the misspelled word `label1led`.

[Click here to view code image](#)

```
45 // matching zero or one occurrences of a sub
46 regex r7("label1?ed");
47 cout << "Matching against: label1?ed\n"
48     << fmt::format("labelled: {}; labeled: {}",
49                     regex_match("labelled", r7), regex_
50                     match("label1led", r7));
```

51

```
< ━━━━ >  
Matching against: labell?ed  
labelled: true; labeled: true; labellled: false
```

You can *match at least n occurrences of a subexpression to its left* with the **{n,}** quantifier. The regular expression in line 53 matches strings containing at least three digits:

[Click here to view code image](#)

```
52 // matching n (3) or more occurrences of a su  
53 regex r8(R"(\d{3,})");  
54 cout << R"(Matching against: \d{3,})" << "\n"  
55 << fmt::format("123: {}; 1234567890: {};  
56             regex_match(\"123\", r8), regex_matc  
57             regex_match(\"12\", r8));  
58
```

```
< ━━━━ >  
Matching against: \d{3,}  
123: true; 1234567890: true; 12: false
```

You can *match between n and m (inclusive) occurrences of a subexpression* with the **{n,m}** quantifier. The regular expression in line 60 matches strings containing 3 to 6 digits:

[Click here to view code image](#)

```
59 // matching n to m inclusive (3-6), occurrenc  
60 regex r9(R"(\d{3,6})");  
61 cout << R"(Matching against: \d{3,6})" << "\n"  
62 << fmt::format("123: {}; 123456: {}; 1234567:  
63             regex_match(\"123\", r9), regex_matc  
64             regex_match(\"1234567\", r9), regex_
```

```
65 }
```

```
Matching against: \d{3,6}\n123: true; 123456: true; 1234567: false; 12: false
```

8.20.2 Replacing Substrings

The header `<regex>` provides function `regex_replace` to replace patterns in a string. Let's convert a tab-delimited string to comma-delimited (Fig. 8.17). The `regex_replace` (line 15) function receives three arguments:

- the string to be searched ("1\t2\t3\t4")
- the regex pattern to match (the tab character "\t") and
- the replacement text (", ")

and returns a new string containing the modifications. The expression `regex{R"(\t)"}` in line 15 creates a temporary `regex` object, initializes it and immediately passes it to `regex_replace`. This is useful if you do not need to reuse a `regex` multiple times.

[Click here to view code image](#)

```
1 // fig08_17.cpp
2 // Regular expression replacements and split
3 #include <iostream>
4 #include <regex>
5 #include <string>
6 #include <vector>
7 #include "fmt/format.h" // In C++20, this will be part of the standard library
8 using namespace std;
9
10 int main() {
11     // replace tabs with commas
12     string s1("1\t2\t3\t4");
```

```
13     cout << "Original string: 1\t2\t3\t4
14     cout << fmt::format("After replacing tabs:
15             regex_replace(s1, regex{R"(
16 })
```

```
<          >
Original string: 1\t2\t3\t4
After replacing tabs with commas: 1,2,3,4
```

Fig. 8.17 Regular expression replacements and splitting.

8.20.3 Searching for Matches

You can match a substring within a string using function `regex_search` (Fig. 8.18), which returns `true` if any part of an arbitrary string matches the regular expression. Optionally, the function also gives you access to the matching substring via an object of the class template `match_results` that you pass as an argument. There are `match_results` aliases for different string types:

- For searches in `std::strings` and `string` object literals, use an `smatch` (pronounced “ess match”).
- For searches on C-strings and `string` literals, you use a `cmatch` (pronounced “see match”).

[Click here to view code image](#)

```
1 // fig08_18.cpp
2 // Matching patterns throughout a string.
3 #include <iostream>
4 #include <regex>
5 #include <string>
6 #include "fmt/format.h" // In C++20, this is
7 using namespace std;
8
9 int main() {
```

```
10 // performing a simple match
11 string s1{"Programming is fun"};
12 cout << fmt::format("s1: {}\n\n", s1);
13 cout << "Search anywhere in s1:\n"
14     << fmt::format("Programming: {}; fun"
15                 regex_search(s1, regex{"Program")
16                 regex_search(s1, regex{"fun"})
17                 regex_search(s1, regex{"fn"}))
18
```

< >

```
s1: Programming is fun

Search anywhere in s1:
Programming: true; fun: true; fn: false
```

Fig. 8.18 Matching patterns throughout a string.

Finding a Match Anywhere in a String

The calls to function `regex_search` in lines 15–17, search in `s1` ("Programming is fun") for the first occurrence of a substring that matches a regular expression—in this case, the literal strings "Programming", "fun" and "fn". Function `regex_search`'s two-argument version simply returns `true` or `false` to indicate a match.

Ignoring Case in a Regular Expression and Viewing the Matching Text

The `regex_constants` in the header `<regex>` can customize how regular expressions perform matches. For example, matches are case sensitive by default, but by using the constant `regex_constants::icase`, you can perform a case-insensitive search.

[Click here to view code image](#)

```
19 // ignoring case
20 string s2{"SAM WHITE"};
21 smatch match; // store the text that matches
```

```
22     cout << fmt::format("s2: {}\n\n", s2);
23     cout << "Case insensitive search for Sam in s
24         << fmt::format("Sam: {}\n",
25             regex_search(s2, match, regex{"Sam")));
26     << fmt::format("Matched text: {}\n\n", m
27
```

```
s2: SAM WHITE
```

```
Case insensitive search for Sam in s2:
```

```
Sam: true
```

```
Matched text: SAM
```

Line 25 calls `regex_search`'s three-argument version in which the arguments are:

- the string to search (`s2`; line 20), which in this case contains all capital letters,
- the `smatch` object (line 21) that stores the match if there is one, and
- the `regex` pattern to match.

Here, we created the `regex` with a second argument

```
regex{"Sam", regex_constants::icase}
```

This `regex` matches the literal characters "Sam" regardless of their case. So, in `s2`, "SAM" matches the `regex` "Sam" because both have the same letters, even though "SAM" contains only uppercase letters. To confirm the matching text, line 26 gets the `match`'s string representation by calling its member function `str`.

Finding All Matches in a String

Let's extract all the U.S. phone numbers of the form # #-# #-# # # from a string. The following code finds each substring in `contact` (line 29) that matches the `regex` `phone` (line 30) and displays the matching text.

[Click here to view code image](#)

```
28     // finding all matches
29     string contact{"Wally White, Home: 555-555-12
30     regex phone{R"(\d{3}-\d{3}-\d{4})"} ;
31
32     cout << "\nFinding all phone numbers in:\n" <
33     while (regex_search(contact, match, phone)) {
34         cout << fmt::format("    {}\\n", match.str())
35         contact = match.suffix();
36     }
37 }
```

```
Finding all phone numbers in:
Wally White, Home: 555-555-1234, Work: 555-555-4321
    555-555-1234
    555-555-4321
```

The `while` statement iterates as long as `regex_search` returns `true`—that is, as long as it finds a match. Each iteration of the loop

- displays the substring that matched the regular expression (line 34), then
- replaces `contact` with the result of calling the `match` object's member function `suffix` (line 35), which returns the portion of the string that has not yet been searched. This new `contact` string is used in the next call to `regex_search`.

8.21 Wrap-Up

This chapter discussed more details of `std::string`, such as assigning, concatenating, comparing, searching and swapping strings. We also introduced several member functions to determine string characteristics, to find, replace and insert characters in a string, and to convert strings to pointer-based strings and vice versa. We performed input from and output to strings in memory, and introduced functions for converting numeric values to strings and for converting strings to numeric values.

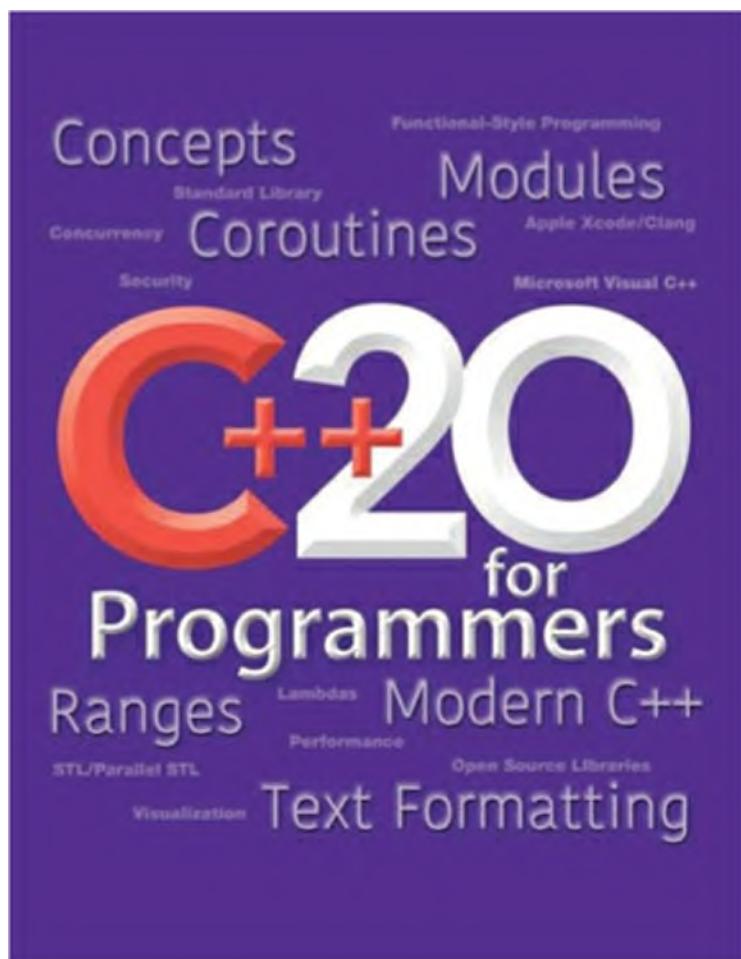
We presented sequential text-file processing using features from the header <fstream> to manipulate persistent data, then demonstrated string stream processing. In our first objects-natural case study, we used an open source library to read the contents of the Titanic dataset from a CSV file, then performed some basic data analytics. In our second objects-natural case study, we introduced regular expressions for pattern matching.

We've now introduced the basic concepts of control statements, functions, arrays, vectors, strings and files. You've already seen in many of our objects natural case studies, that C++ applications typically create and manipulate objects that perform the work of the application.

In [Chapter 9](#), you'll learn how to implement your own custom classes and use objects of those classes in applications. We'll begin discussing class-design and related software-engineering concepts.

Part 3: Object-Oriented Programming

Chapter 9. Custom Classes



Objectives

In this chapter, you'll:

- Define a custom class and use it to create objects.
- Implement a class's behaviors as member functions and attributes as data members.
- Access and manipulate private data members through public *get* and *set* functions to enforce data encapsulation.
- Use a constructor to initialize an object's data.

- Separate a class's interface from its implementation for reuse.
 - Access class members via the dot (.) and arrow (->) operators.
 - Use destructors to perform “termination housekeeping.”
 - Learn why returning a reference or a pointer to `private` data is dangerous.
 - Assign the data members of one object to those of another.
 - Create objects composed of other objects.
 - Use `friend` functions and declare `friend` classes.
 - Access non-`static` class members via the `this` pointer.
 - Use `static` data members and member functions.
 - Use `structs` to create aggregate types, and use C++20 designated initializers to initialize aggregate members.
 - Do an objects-natural case study that serializes objects using JavaScript Object Notation (JSON) and the `cereal` library.
-

Outline

[9.1 Introduction](#)

[9.2 Test-Driving an Account Object](#)

[9.3 Account Class with a Data Member and *Set* and *Get* Member Functions](#)

[9.3.1 Class Definition](#)

[9.3.2 Access Specifiers `private` and `public`](#)

[9.4 Account Class: Custom Constructors](#)

[9.5 Software Engineering with *Set* and *Get* Member Functions](#)

[9.6 Account Class with a Balance](#)

[9.7 Time Class Case Study: Separating Interface from Implementation](#)

[9.7.1 Interface of a Class](#)

[9.7.2 Separating the Interface from the Implementation](#)

[9.7.3 Class Definition](#)

- 9.7.4 Member Functions**
- 9.7.5 Including the Class Header in the Source-Code File**
- 9.7.6 Scope Resolution Operator (::)**
- 9.7.7 Member Function setTime and Throwing Exceptions**
- 9.7.8 Member Function to24HourString**
- 9.7.9 Member Function to12HourString**
- 9.7.10 Implicitly Inlining Member Functions**
- 9.7.11 Member Functions vs. Global Functions**
- 9.7.12 Using Class Time**
- 9.7.13 Object Size**
- 9.8 Compilation and Linking Process**
- 9.9 Class Scope and Accessing Class Members**
- 9.10 Access Functions and Utility Functions**
- 9.11 Time Class Case Study: Constructors with Default Arguments**
 - 9.11.1 Class Time**
 - 9.11.2 Overloaded Constructors and C++11 Delegating Constructors**
- 9.12 Destructors**
- 9.13 When Constructors and Destructors Are Called**
- 9.14 Time Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a private Data Member**
- 9.15 Default Assignment Operator**
- 9.16 const Objects and const Member Functions**
- 9.17 Composition: Objects as Members of Classes**
- 9.18 friend Functions and friend Classes**
- 9.19 The this Pointer**
 - 9.19.1 Implicitly and Explicitly Using the this Pointer to Access an Object's Data Members**
 - 9.19.2 Using the this Pointer to Enable Cascaded Function Calls**
- 9.20 static Class Members—Classwide Data and Member Functions**
- 9.21 Aggregates in C++20**
 - 9.21.1 Initializing an Aggregate**
 - 9.21.2 C++20: Designated Initializers**

9.22 Objects Natural Case Study: Serialization with JSON

9.22.1 Serializing a vector of Objects containing public Data

9.22.2 Serializing a vector of Objects containing private Data

9.23 Wrap-Up

9.1 Introduction¹

1. This chapter depends on the terminology and concepts introduced in Section 1.16, Introduction to Object Orientation and “Objects Natural.”

Section 1.16 presented a friendly introduction to object orientation, discussing classes, objects, data members (attributes) and member functions (behaviors). In our objects-natural case studies, you’ve created objects of existing classes and called their member functions to make the objects perform powerful tasks without having to know how these classes worked internally.

This chapter begins our deeper treatment of object-oriented programming as we craft valuable custom classes. C++ is an **extensible programming language**—each class you create becomes a new type you can use to create objects. Some development teams in industry work on applications that contain hundreds, or even thousands, of custom classes.

9.2 Test-Driving an Account Object

We begin our introduction to custom classes with three examples that create an `Account` class representing a simple bank account. First, let’s look at the `main` program and output, so you can see an object of our initial `Account` class in action. To help you prepare for the larger programs you’ll encounter later in this book and in industry, we define the `Account` class and `main` in separate files—`main` in `AccountTest.cpp` (Fig. 9.1) and class `Account` in `Account.h`, which we’ll show in Fig. 9.2.

[Click here to view code image](#)

¹ // Fig. 9.1: `AccountTest.cpp`

```
2 // Creating and manipulating an Account obj
3 #include <iostream>
4 #include <string>
5 #include <fmt/format.h> // In C++20, this w
6 #include "Account.h"
7
8 using namespace std;
9
10 int main() {
11     Account myAccount{}; // create Account ob
12
13     // show that the initial value of myAcco
14     cout << fmt::format("Initial account nam
15
16     // prompt for and read the name
17     cout << "Enter the account name: ";
18     string name{};
19     getline(cin, name); // read a line of te
20     myAccount.setName(name); // put name in
21
22     // display the name stored in object myA
23     cout << fmt::format("Updated account nam
24 }
```

< >

```
Initial account name:
Enter the account name: Jane Green
Updated account name: Jane Green
```

Fig. 9.1 Creating and manipulating an Account object.

[Click here to view code image](#)

```
1 // Fig. 9.2: Account.h
2 // Account class with a data member and
3 // member functions to set and get its val
```

```

4  #include <string>
5  #include <string_view>
6
7  class Account {
8  public:
9      // member function that sets m_name in t
10     void setName(std::string_view name) {
11         m_name = name; // replace m_name's va
12     }
13
14     // member function that retrieves the ac
15     const std::string& getName() const {
16         return m_name; // return m_name's val
17     }
18 private:
19     std::string m_name; // data member conta
20 }; // end class Account

```



Fig. 9.2 Account class with a data member and member functions to *set* and *get* its value.

Instantiating an Object

Typically, you cannot call a class's member functions until you create an object of that class.² Line 11

2. In [Section 9.20](#), you'll see that `static` member functions are an exception.

[Click here to view code image](#)

```
Account myAccount{}; // create Account object myAc
```



creates an object called `myAccount`. The variable's type is `Account`—the class we'll define in [Fig. 9.2](#).

Headers and Source-Code Files

When we declare `int` variables, the compiler knows what `int` is—it's a fundamental type that's built into C++. In line 11, however, the compiler

does not know in advance what type Account is—it's a **user-defined type**.

When packaged properly, new classes can be reused by other programmers. It's customary to place a reusable class definition in a file known as a **header** with a .h filename extension.³ You include that header wherever you need to use the class, as you've been doing throughout this book with C++ Standard Library and third-party library classes.

3. C++ Standard Library headers, like <iostream>, do not use the .h filename extension and some C++ programmers prefer the .hpp extension.

We tell the compiler what an Account is by including its header, as in line 6:

```
#include "Account.h"
```

If we omit this, the compiler issues error messages wherever we use class Account and any of its capabilities. A header that you define in your program is placed in double quotes (" "), rather than the angle brackets (<>). The double quotes tell the compiler to check the folder containing AccountTest.cpp (Fig. 9.1) before the compiler's header search path.

Calling Class Account's getName Member Function

Class Account's getName member function returns the name stored in a particular Account object. Line 14 calls myAccount.getName() to get the myAccount object's initial name, which is the empty string. We'll say more about this shortly.

Calling Class Account's setName Member Function

The setName member function stores a name in a particular Account object. Line 20 calls

myAccount's setName member function to store name's value in the object myAccount.

Displaying the Name That Was Entered by the User

To confirm that myAccount now contains the name you entered, line 23 calls member function getName again and displays its result.

9.3 Account Class with a Data Member and Set and

Get Member Functions

Now that we've seen class `Account` in action (Fig. 9.1), we present its internal details.

9.3.1 Class Definition

Class `Account` (Fig. 9.2) contains an `m_name` data member (line 19) that stores the account holder's name. Each object of the class has its own copy of the class's data members.⁴ Later, we'll add a `balance` data member to keep track of the money in each `Account`. Class `Account` also contains:

4. In Section 9.20, you'll see that `static` data members are an exception.
- member function `setName` (lines 10–12) that stores a name in an `Account`, and
- member function `getName` (lines 15–17) that retrieves a name from an `Account`.

Keyword `class` and the Class Body

The class definition begins with the keyword `class` (line 7) followed immediately by the class's name—in this case, `Account`. By convention:

- each word in a class name starts with a capital first letter, and
- data-member and member-function names begin with a lowercase first letter.

Every class's body is enclosed in braces `{ }` (lines 7 and 20). The class definition terminates with a required semicolon (line 20).

Data Member `m_name` of Type `std::string`

Recall from Section 1.16 that an object has attributes, implemented as data members. Each object maintains its own copy of these throughout its lifetime. Usually, a class also contains member functions that manipulate the data members of particular objects of the class. The data members exist:

- before a program calls member functions on a specific object,
- while the member functions are executing and
- after the member functions finish executing.

Data members are declared inside a class definition but outside the class's member functions. Line 19

[Click here to view code image](#)

```
std::string m_name; // data member containing account name
```

declares a `string` data member called `m_name`. The "`m_`" prefix is a common naming convention to indicate that a variable represents a data member. If there are many `Account` objects, each has its own `m_name`. Because `m_name` is a data member, it can be manipulated by the class's member functions. Recall that the default `string` value is the empty string (""), which is why line 14 in `main` (Fig. 9.1) did not display a name.

By convention, C++ programmers typically place a class's data members last in the class's body. You can declare the data members anywhere in the class outside its member-function definitions, but scattering the data members can lead to hard-to-read code.

Use `std::` with Standard Library Components in Headers

Throughout the `Account.h` header (Fig. 9.2), we use `std::` when referring to `string` (lines 10, 15 and 19). For subtle reasons that we explain in Section 19.4, headers should not contain using directives or using declarations.

`setName` Member Function

Member function `setName` (lines 10–12):

[Click here to view code image](#)

```
void setName(std::string_view name) {
    m_name = name; // replace m_name's value with name
```

 SE receives a `string_view` representing the `Account`'s name and assigns the `name` argument to data member `m_name`. Recall that a `string_view` is a read-only view into a character sequence, such as a

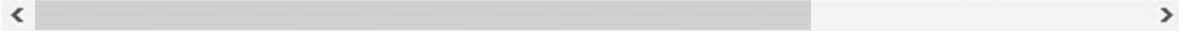
`std::string` or a C-string. This copies `name`'s characters into `m_name`. The "`m_`" in `m_name` makes it easy to distinguish the parameter name from the data member `m_name`.

getName Member Function

Member function `getName` (lines 15–17):

[Click here to view code image](#)

```
const std::string& getName() const {  
    return m_name; // return m_name's value to this  
}
```



has no parameters and returns a particular `Account` object's `m_name` to the caller as a `const std::string&`. Declaring the returned reference `const` ensures that the caller cannot modify the object's data via that reference.

const Member Functions

Note the `const` to the right of the parameter list in `getName`'s header (line 15). When returning `m_name`, member function `getName` does not, and should not, modify the `Account` object on which it's called. Declaring a member function `const` tells the compiler, "this function should not modify the object on which it's called—if it does, please issue a compilation error." This can help you locate errors if you accidentally insert code that would modify the object. It also tells the compiler that `getName` may be called on a `const Account` object, or called via a reference or pointer to a `const Account`.

9.3.2 Access Specifiers `private` and `public`

 **CG** The keyword `private` (line 18) is an **access specifier**. Each access specifier is followed by a required colon (:). Data member `m_name`'s declaration (line 19) appears after `private:` to indicate that `m_name` is accessible only to class `Account`'s member functions.⁵ This is known as **information hiding** (or, more generally, as hiding implementation details) and is a recommended practice of the C++ Core Guidelines⁶ and object-

oriented programming in general. The data member `m_name` is encapsulated (hidden) and can be used only in class `Account`'s `setName` and `getName` member functions. Most data-member declarations appear after the `private:` access specifier. We generally omit the colon when we refer to the `private` and `public` access specifiers in the text, as we did in this sentence.

5. Or to “friends” of the class, as you’ll see in [Section 9.18](#).
6. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-private>.

This class also contains the `public` access specifier (line 8):

```
public:
```

 **SE** Data members or member functions listed after access specifier `public`—and before the next access specifier if there is one—are “available to the public.” They can be used anywhere an object of the class is in scope. Making a class’s data members `private` facilitates debugging because problems with data manipulations are localized to the class’s member functions. In [Chapter 10](#), we’ll introduce the `protected` access specifier.

Default Access for Class Members

By default, everything in a class is `private`, unless you specify otherwise. Once you list an access specifier, everything has that level of access until you list another access specifier. The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing. We prefer to list `public` only once, grouping everything that’s `public`. We prefer to list `private` only once, grouping everything that’s `private`.

9.4 Account Class: Custom Constructors

As you saw in the preceding example, when an `Account` object is created, its `string` data member `m_name` is initialized to the empty `string` by default. But what if you want to provide a name when you first create an `Account` object? Each class can define `constructors` that specify custom initialization for objects of that class. A constructor is a special member

function that must have the same name as the class. Constructors cannot return values, so they do not specify a return type (not even `void`). C++ guarantees that a constructor is called when each object is created, so this is the ideal point to initialize an object's data members. Every time you created an object so far in this book, the corresponding class's constructor was called to initialize the object. As you'll soon see, classes may have overloaded constructors.

Like member functions, constructors can have parameters—the corresponding argument values help initialize the object's data members. For example, you can specify an `Account` object's name when the object is created, as you'll do in line 12 of [Fig. 9.4](#):

[Click here to view code image](#)

```
Account account1{"Jane Green"};
```

In the preceding statement, the string "Jane Green" is passed to the `Account` class's constructor and used to initialize the `account1` object's data. The preceding statement assumes that class `Account` has a constructor that can receive a string argument.

Account Class Definition

[Figure 9.3](#) shows class `Account` with a constructor that receives an `accountName` parameter and uses it to initialize the data member `m_name` when an `Account` object is created.

[Click here to view code image](#)

```
1 // Fig. 9.3: Account.h
2 // Account class with a constructor that initializes data member m_name
3 #include <string>
4 #include <string_view>
5
6 class Account {
7 public:
8     // constructor initializes data member m_name
9     explicit Account(std::string_view name)
```

```
10         : m_name{name} { // member initialize
11             // empty body
12         }
13
14     // function to set the account name
15     void setName(std::string_view name) {
16         m_name = name; // replace m_name's va
17     }
18
19     // function to retrieve the account name
20     const std::string& getName() const {
21         return m_name;
22     }
23 private:
24     std::string m_name; // account name data
25 }; // end class Account
```



Fig. 9.3 Account class with a constructor that initializes the account name.

Account Class's Custom Constructor Definition

Lines 9–12 of Fig. 9.3 define Account's constructor:

[Click here to view code image](#)

```
explicit Account(std::string_view name)
: m_name{name} { // member initializer
// empty body }
```

Usually, constructors are `public`, so any code with access to the class definition can create and initialize objects of that class.⁷ Line 9 indicates that the constructor has a `name` parameter of type `string_view`. When you create a new `Account` object, you must pass a person's name to the constructor, which then initializes the data member `m_name` with the contents of the `string_view` parameter.

7. Section 11.10.2 discusses why you might use a `private` constructor.

The constructor's **member-initializer list** (line 10):

```
: m_name{ name }
```

initializes the `m_name` data member. Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body. You separate the member initializer list from the parameter list with a colon (:).

 **SE** Each member initializer consists of a data member's variable name followed by braces containing its initial value.⁸ This member initializer calls the `std::string` class's constructor that receives a `string_view`. If a class has more than one data member, each member initializer is separated from the next by a comma. Member initializers execute in the order you declare the data members in the class so, for clarity, list the member initializers in the same order. The member initializer list executes before the constructor's body executes.

⁸. Occasionally, parentheses rather than braces may be required, such as when initializing a `vector` of a specified size, as we did in Fig. 6.14.

 **CG**  **PERF** Though you can perform initialization with assignment statements in the constructor's body, the C++ Core Guidelines recommend using member initializers.⁹ You'll see later that member initializers can be more efficient. Also, you'll see that certain data members must be initialized using the member-initializer syntax, because you cannot assign to them in the constructor's body.

⁹. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-initialize>.

explicit Keyword

 **CG** A constructor with only one parameter should be declared **explicit** to prevent the compiler from using the constructor to perform implicit type conversions.¹⁰ The keyword `explicit` means that `Account`'s constructor must be called explicitly, as in:

¹⁰. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc->

`explicit`.

[Click here to view code image](#)

```
Account account1{"Jane Green"};
```

 **ERR** For now, simply declare all single-parameter constructors `explicit`. In Section 11.13, you'll see that single-parameter constructors without `explicit` can be called implicitly to perform type conversions. Such implicit constructor calls can lead to subtle errors and are generally discouraged. Line 9 of Fig. 9.3 does not specify a return type (not even `void`) because, again, constructors cannot return values. Also, constructors cannot be declared `const` (simply because initializing an object modifies it).

Initializing Account Objects When They're Created

The `AccountTest` program (Fig. 9.4) initializes two `Account` objects using the constructor. Line 12 creates the `Account` object `account1`:

[Click here to view code image](#)

```
Account account1{"Jane Green"};
```

[Click here to view code image](#)

```
1 // Fig. 9.4: AccountTest.cpp
2 // Using the Account constructor to initialize
3 // member at the time each Account object is created
4 #include <iostream>
5 #include <fmt/format.h> // In C++20, this will be <format>
6 #include "Account.h"
7
8 using namespace std;
9
10 int main() {
11     // create two Account objects
12     Account account1{"Jane Green"};
13     Account account2{"John Blue"};
14 }
```

```
15     // display initial each Account's corres;
16     cout << fmt::format("account1 name is: {"
17             account1.getName(), account2.
18 }
```

```
<          >
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 9.4 Using the Account constructor to initialize the `m_name` data member at the time each Account object is created.

When you create an object, C++ calls the class's constructor to initialize that object. In line 12, the argument "Jane Green" is used by the constructor to initialize the new object's `m_name` data member, as specified in lines 9–12 of Fig. 9.3. Line 13 of Fig. 9.4 repeats this process, passing the argument "John Blue" to initialize `m_name` for `account2`:

[Click here to view code image](#)

```
Account account2{"John Blue"};
```

To confirm the objects were initialized properly, lines 16–17 call the `getName` member function to get each object's name. The output shows different names, confirming that each `Account` maintains its own copy of the class's data member `m_name`.

Default Constructor

Recall that line 11 of Fig. 9.1 created an `Account` object with empty braces to the right of the object's variable name:

```
Account myAccount{};
```

which, for an object of a class, is equivalent to:

```
Account myAccount;
```

In the preceding statements, C++ implicitly calls `Account`'s **default constructor**. In any class that does not define a constructor, the compiler

generates a default constructor with no parameters. The default constructor does not initialize the class's fundamental-type data members but does call the default constructor for each data member that's an object of another class. For example, though you do not see this in the first Account class's code (Fig. 9.2), Account's default constructor calls class `std::string`'s default constructor to initialize the data member `m_name` to the empty string (""). An uninitialized fundamental-type variable contains an undefined ("garbage") value.

There's No Default Constructor in a Class That Defines a Constructor

 **SE** If you define a custom constructor for a class, the compiler will not create a default constructor for that class. In that case, you will not be able to create an `Account` object by calling the constructor with no arguments unless the custom constructor you define has an empty parameter list or has default arguments for all its parameters. We'll show later that you can force the compiler to create the default constructor even if you've defined non-default constructors. Unless default initialization of your class's data members is acceptable, provide a custom constructor that initializes each new object's data members with meaningful values.

C++'s Special Member Functions

In addition to the default constructor, the compiler can generate default versions of five other **special member functions**—a copy constructor, a move constructor, a copy assignment operator, a move assignment operator and a destructor. We'll briefly introduce copy construction, copy assignment and destructors in this chapter. In [Chapter 11](#), we'll discuss the details of all six special member functions, including:

- when they're auto-generated,
- when you might need to define custom versions of each, and
- the various C++ Core Guidelines for these special member functions.

You'll see that you should try to construct your custom classes such that the compiler can auto-generate these special member functions for you—this is called the "Rule of Zero."

9.5 Software Engineering with *Set* and *Get* Member

Functions

As you'll see in the next section, *set* and *get* member functions can validate attempts to modify private data and control how that data is presented to the caller, respectively. These are compelling software engineering benefits.

A **client** of the class is any other code that calls the class's member functions. If a data member were `public`, any client could see the data and do whatever it wanted with it, including setting it to an invalid value.

You might think that even though a client cannot directly access a `private` data member, the client can nevertheless do whatever it wants with the variable through `public` *set* and *get* functions. You'd think that you could peek at the `private` data (and see exactly how it's stored in the object) anytime with the `public` *get* function and that you could modify the `private` data at will through the `public` *set* function.

 **SE** Actually, *set* functions can be written to validate their arguments and reject any attempts to *set* the data to incorrect values, such as

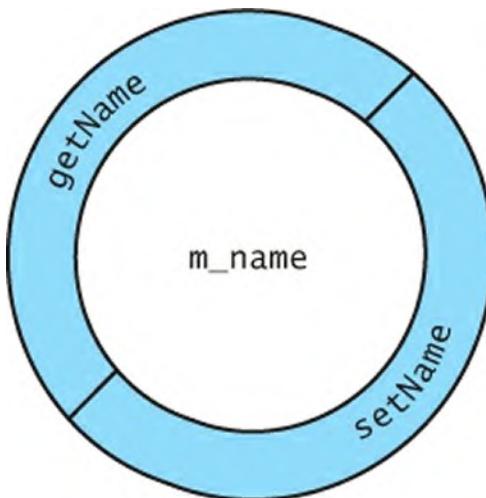
- a negative body temperature
- a day in March outside the range 1 through 31, or
- a product code not in the company's product catalog.

 **SE**  **SE** A *get* function can present the data in a different form, keeping the object's actual data representation hidden from the user. For example, a `Grade` class might store a numeric grade as an `int` between 0 and 100, but a `getGrade` member function might return a letter grade as a `string`, such as "A" for grades between 90 and 100, "B" for grades between 80 and 89, etc. Tightly controlling the access to and presentation of `private` data can reduce errors while increasing your programs' robustness, security and usability.

Conceptual View of an Account Object with Encapsulated Data

You can think of an `Account` object, as shown in the following diagram. The `private` data member `m_name`, represented by the inner circle, is hidden inside the object and accessible only via an outer layer of `public` member functions, represented by the outer ring containing `getName` and

`setName`. Any client that needs to interact with the `Account` object can do so only by calling the `public` member functions of the outer layer.



 **SE** Generally, data members are `private`, and the member functions that a class's clients need to use are `public`. Later, we'll discuss why you might use a `public` data member or a `private` member function. Using `public` *set* and *get* functions to control access to `private` data makes programs clearer and easier to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified, and possibly often.

9.6 Account Class with a Balance

 **CG** Figure 9.5 defines an `Account` class that maintains two related pieces of data—a bank account's balance and the account holder's name. The C++ Core Guidelines recommend defining related data items in a class (or, as you'll see in Section 9.21, a `struct`).¹¹

¹¹. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-org>.

[Click here to view code image](#)

```
1 // Fig. 9.5: Account.h
2 // Account class with m_name and m_balance
3 // constructor and deposit function that ea
4 #include <string>
5 #include <string_view>
6
7 class Account {
8 public:
9     // Account constructor with two parameters
10    Account(std::string_view name, double ba
11             : m_name{name} { // member initialize
12
13        // validate that balance is greater than zero
14        // data member m_balance keeps its value
15        if (balance > 0.0) { // if the balance is
16            m_balance = balance; // assign it
17        }
18    }
19
20    // function that deposits (adds) only a positive amount
21    void deposit(double amount) {
22        if (amount > 0.0) { // if the amount is positive
23            m_balance += amount; // add it to the balance
24        }
25    }
26
27    // function returns the account balance
28    double getBalance() const {
29        return m_balance;
30    }
31
32    // function that sets the account name
33    void setName(std::string_view name) {
34        m_name = name; // replace m_name's value
35    }
36
37    // function that returns the account name
```

```

38     const std::string& getName() const {
39         return m_name;
40     }
41 private:
42     std::string m_name; // account name data
43     double m_balance{0.0}; // data member wi-
44 }; // end class Account

```



Fig. 9.5 Account class with `m_name` and `m_balance` data members, and a constructor and `deposit` function that each perform validation.

Data Member `balance`

A typical bank services many accounts, each with its own balance. Every object of this updated `Account` class contains its own copies of data members `m_name` and `m_balance`. Line 43 declares a `double` data member `m_balance` and initializes its value to `0.0`:

[Click here to view code image](#)

```
double m_balance{0.0}; // data member with default
```



CG CG ⚡ This is an **in-class initializer**. The C++ Core Guidelines recommend using in-class initializers when a data member should be initialized with a constant.¹² The Core Guidelines also recommend when possible that you initialize all your data members with in-class initializers and let the compiler generate a default constructor for your class. A compiler-generated default constructor can be more efficient than one you define.¹³ Like `m_name`, we can use `m_balance` in the class's member functions (lines 16, 23 and 29) because it's a data member of the class.

12. C++ Core Guidelines. Accessed July 12, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-in-class-initializer>.

13. C++ Core Guidelines. Accessed July 12, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default>.

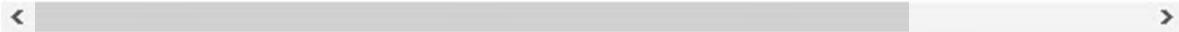
Two-Parameter Constructor

The class has a constructor and four member functions. It's common for someone opening an account to deposit money immediately, so the constructor (lines 10–18) receives a second parameter `balance` of type `double` that represents the starting balance. We did not declare this constructor `explicit`, because it cannot be called with only one parameter.

Lines 15–17 ensure that data member `m_balance` is assigned the `balance` parameter's value only if that value is greater than `0.0`:

[Click here to view code image](#)

```
if (balance > 0.0) { // if the balance is valid  
    m_balance = balance; // assign it to data member  
}
```



Otherwise, `m_balance` will still have its default initial value `0.0` that was set at line 43 in class `Account`'s definition.

deposit Member Function

Member function `deposit` (lines 21–25) receives a `double` parameter `amount` and does not return a value. Lines 22–24 ensure that parameter `amount`'s value is added to `m_balance` only if `amount` is greater than zero (that is, it's a valid deposit amount).

getBalance Member Function

Member function `getBalance` (lines 28–30) allows the class's clients to obtain a particular `Account` object's `m_balance` value. The member function specifies return type `double` and an empty parameter list. Like member function `getName`, `getBalance` is declared `const` because, in the process of returning `m_balance`, the function does not, and should not, modify the `Account` object on which it's called.

Manipulating Account Objects with Balances

The main function in Fig. 9.6 creates two `Account` objects (lines 10–11) and attempts to initialize them with a valid balance of `50.00` and an invalid balance of `-7.00`, respectively. For our examples, we assume that balances

must be greater than or equal to zero. Lines 14–17 output both Accounts' names and balances by calling each objects' `getName` and `getBalance` member functions. The `account2` object's balance is initially 0.0 because the constructor rejected the attempt to start `account2` with a negative balance, so `account2`'s `m_balance` data member retains its default initial value.

[Click here to view code image](#)

```
1 // Fig. 9.6: AccountTest.cpp
2 // Displaying and updating Account balances
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this w
5 #include "Account.h"
6
7 using namespace std;
8
9 int main() {
10    Account account1{"Jane Green", 50.00};
11    Account account2{"John Blue", -7.00};
12
13    // display initial balance of each objec
14    cout << fmt::format("account1: {} balanc
15                account1.getName(), account1.
16    cout << fmt::format("account2: {} balanc
17                account2.getName(), account2.
```



```
account1: Jane Green balance is $50.00
account2: John Blue balance is $0.00
```

Fig. 9.6 Displaying and updating Account balances.

Reading a Deposit Amount from the User and Making a Deposit

Lines 19–22 prompt for, input and display the account1 deposit amount. Line 23 calls object account1’s deposit member function with variable amount as the argument to add that value to account1’s balance. Lines 26–29 (Fig. 9.6) output both Accounts’ names and balances again to show that only account1’s balance has changed.

[Click here to view code image](#)

```
19      cout << "Enter deposit amount for account1:  
20      double amount;  
21      cin >> amount; // obtain user input  
22      cout << fmt::format("adding ${:.2f} to acco  
23      account1.deposit(amount); // add to account  
24  
25      // display balances  
26      cout << fmt::format("account1: {} balance i  
27                  account1.getName(), account1.get  
28      cout << fmt::format("account2: {} balance i  
29                  account2.getName(), account2.get  
30
```

```
Enter deposit amount for account1: 25.37  
adding $25.37 to account1 balance
```

```
account1: Jane Green balance is $75.37  
account2: John Blue balance is $0.00
```

Lines 31–33 prompt for, input and display account2’s deposit amount. Line 34 calls object account2’s deposit member function with variable amount as the argument to add that value to account2’s balance. Finally, lines 37–40 output both Accounts’ names and balances again to show that only account2’s balance has changed.

[Click here to view code image](#)

```
31      cout << "Enter deposit amount for account2:
```

```
32     cin >> amount; // obtain user input
33     cout << fmt::format("adding ${:.2f} to acco
34     account2.deposit(amount); // add to account
35
36     // display balances
37     cout << fmt::format("account1: {} balance i
38                 account1.getName(), account1.get
39     cout << fmt::format("account2: {} balance i
40                 account2.getName(), account2.get
41 }
```

< >

```
Enter deposit amount for account2: 123.45
adding $123.45 to account2 balance
```

```
account1: Jane Green balance is $75.37
account2: John Blue balance is $123.45
```

9.7 Time Class Case Study: Separating Interface from Implementation

Each of our prior custom class definitions placed a class in a header for reuse, then included the header into a source-code file containing `main`, so we could create and manipulate objects of the class. Unfortunately, placing a complete class definition in a header reveals the class's entire implementation to its clients. A header is simply a text file that anyone can open and read.

 **SE** Conventional software-engineering wisdom says that to use an object of a class, the client code (e.g., `main`) needs to know only

- what member functions to call,
- what arguments to provide to each member function and
- what return type to expect from each member function.

The client code does not need to know how those functions are implemented. This is another example of the principle of least privilege.

If the client-code programmer knows how a class is implemented, the programmer might write client code based on the class's implementation details. Ideally, if that implementation changes, the class's clients should not have to change. Hiding the class's implementation details makes it easier to change the implementation while minimizing, and hopefully eliminating, changes to client code.

Our next example creates and manipulates an object of class `Time`.¹⁴ We demonstrate two important C++ software engineering concepts:

14. In professional C++ development, rather than building your own classes to represent times and dates, you'll typically use the header `<chrono>` (<https://en.cppreference.com/w/cpp/chrono>) from the C++ Standard Library.

- **Separating interface from implementation.**
 - Using the preprocessor directive “`#pragma once`” in a header to prevent the header code from being included into the same source code file more than once. Since a class can be defined only once, using such a preprocessing directive prevents multiple-definition errors.

20 C++20 Modules Change How You Separate Interface from Implementation

 **MOD** As you'll see in [Chapter 15](#), C++20 modules¹⁵ eliminate the need for preprocessor `#pragma once`. You'll also see that modules enable you to separate interface from implementation in a single source-code file or by using multiple source-code files.

15. At the time of this writing, the C++20 modules features were not fully implemented by the three compilers we use, so we cover them in a separate chapter.

9.7.1 Interface of a Class

Interfaces define and standardize how things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands. The interface specifies *what* operations a radio permits users to perform but does

not specify *how* the operations are implemented inside the radio.

Similarly, the **interface of a class** describes *what* services a class's clients can use and how to request those services, but not *how* the class implements them. A class's public interface consists of the class's public member functions (also known as the class's **public services**). As you'll soon see, you can specify a class's interface by writing a class definition that lists only the class's member-function prototypes in the class's public section.

9.7.2 Separating the Interface from the Implementation

To separate the class's interface from its implementation, we break up class Time into two files—Time.h (Fig. 9.7) in which class Time is defined and Time.cpp (Fig. 9.8) in which Time's member functions are defined. This split:

1. helps make the class reusable,
2. ensures that the clients of the class know what member functions the class provides, how to call them and what return types to expect, and
3. enables the clients to ignore how the class's member functions are implemented.

[Click here to view code image](#)

```
1 // Fig. 9.7: Time.h
2 // Time class definition.
3 // Member functions are defined in Time.cpp
4 #pragma once // prevent multiple inclusions
5 #include <string>
6
7 // Time class definition
8 class Time {
9 public:
10    void setTime(int hour, int minute, int s)
11    std::string to24HourString() const; // 2
12    std::string to12HourString() const; // 1
13 private:
14    int m_hour{0}; // 0 - 23 (24-hour clock)
```

```
15     int m_minute{0}; // 0 - 59
16     int m_second{0}; // 0 - 59
17 };
```



Fig. 9.7 Time class definition.

[Click here to view code image](#)

```
1 // Fig. 9.8: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept> // for invalid_argument
4 #include <string>
5 #include <fmt/format.h> // In C++20, this will
6 #include "Time.h" // include definition of
7
8 using namespace std;
9
10 // set new Time value using 24-hour time
11 void Time::setTime(int hour, int minute, int second)
12     // validate hour, minute and second
13     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
14         (second < 0 || second >= 60)) {
15     throw invalid_argument{"hour, minute or second is invalid"};
16 }
17
18     m_hour = hour;
19     m_minute = minute;
20     m_second = second;
21 }
22
23 // return Time as a string in 24-hour format
24 string Time::to24HourString() const {
25     return fmt::format("{:02d}:{:02d}:{:02d}")
26 }
27
```

```
28 // return Time as string in 12-hour format
29 string Time::to12HourString() const {
30     return fmt::format("{}:{}{:02d}:{}{:02d} {}"
31             ((m_hour % 12 == 0) ? 12 : m_h
32             (m_hour < 12 ? "AM" : "PM"));
33 }
```



Fig. 9.8 Time class member-function definitions.

 **PERF** In addition, this split can reduce compilation time because the implementation file can be compiled, then does not need to be recompiled unless the implementation changes.

By convention, member-function definitions are placed in a .cpp file with the same base name (e.g., Time) as the class's header. Some compilers support other filename extensions as well. [Figure 9.9](#) defines function main, which is the client of our Time class.

[Click here to view code image](#)

```
1 // fig09_09.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with Ti
4 #include <iostream>
5 #include <stdexcept> // invalid_argument ex
6 #include <string_view>
7 #include <fmt/format.h> // In C++20, this w
8 #include "Time.h" // definition of class Ti
9 using namespace std;
10
11 // displays a Time in 24-hour and 12-hour f
12 void displayTime(string_view message, const
13     cout << fmt::format("{}\n24-hour time: {"
14             message, time.to24HourString(
15
16 }
```

```
17 int main() {
18     Time t{}; // instantiate object t of class Time
19
20     displayTime("Initial time:", t); // display initial time
21     t.setTime(13, 27, 6); // change time
22     displayTime("After setTime:", t); // display changed time
23
24     // attempt to set the time with invalid values
25     try {
26         t.setTime(99, 99, 99); // all values invalid
27     }
28     catch (const invalid_argument& e) {
29         cout << fmt::format("Exception: {}\\n", e.what());
30     }
31
32     // display t's value after attempting to set invalid time
33     displayTime("After attempting to set an invalid time:", t);
34 }
```

```
<          >

Initial time:
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

After setTime:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM
```

Fig. 9.9 Program to test class Time.

9.7.3 Class Definition

The header `Time.h` (Fig. 9.7) contains `Time`'s class definition (lines 8–17). Rather than function definitions, the class contains function prototypes (lines 10–12) that describe the class's public interface without revealing the member-function implementations. The function prototype in line 10 indicates that `setTime` requires three `int` parameters and returns `void`. The prototypes for `to24HourString` and `to12HourString` (lines 11–12) specify that they take no arguments and return a `string`. Classes with one or more constructors would also declare them in the header, as we'll do in subsequent examples.

11 The header still specifies the class's private data members (lines 14–16). Each uses a C++11 in-class initializer to set the data member to 0. The compiler must know the class's data members to determine how much memory to reserve for each object of the class. Including the header `Time.h` in the client code provides the compiler with the information it needs to ensure that the client code calls class `Time`'s member functions correctly.

`#pragma once`

 In larger programs, headers also will contain other definitions and declarations. Attempts to include a header multiple times (inadvertently) often occur in programs with many headers that may themselves include other headers. This could lead to compilation errors if the same definition appears more than once in a preprocessed file. The `#pragma once` directive (line 4) prevents `time.h`'s contents from being included into the same source-code file more than once. In Chapter 15, we'll discuss how C++20 modules help prevent such problems.

9.7.4 Member Functions

`Time.cpp` (Fig. 9.8) defines class `Time`'s member functions, which were declared in lines 10–12 of Fig. 9.7. For member functions `to24HourString` and `to12HourString`, the `const` keyword must appear in both the function prototypes (Fig. 9.7, lines 11–12) and the function definitions (Fig. 9.8, lines 24 and 29).

9.7.5 Including the Class Header in the Source-Code File

To indicate that the member functions in `Time.cpp` are part of class `Time`, we must first include the `Time.h` header (Fig. 9.8, line 6). This allows us to use the class name `Time` in the `Time.cpp` file (lines 11, 24 and 29). When compiling `Time.cpp`, the compiler uses the information in `Time.h` to ensure that

- the first line of each member function matches its prototype in `Time.h`, and
- each member function knows about the class's data members and other member functions.

9.7.6 Scope Resolution Operator (`::`)

Each member function's name (lines 11, 24 and 29) is preceded by the class name and the scope resolution operator (`::`). This “ties” them to the (now separate) `Time` class definition (Fig. 9.7), which declares the class's members. The `Time::` tells the compiler that each member function is in that **class's scope**, and its name is known to other class members.

Without “`Time::`” preceding each function name, the compiler would treat these as global functions with no relationship to class `time`. Such functions, also called “**free**” functions, cannot access `Time`'s private data and cannot call the class's member functions without specifying an object. So, the compiler would not be able to compile these functions because it would not know that class `Time` declares variables `m_hour`, `m_minute` and `m_second`. In Fig. 9.8, lines 18–20, 25 and 31–32 would cause compilation errors because `m_hour`, `m_minute` and `m_second` are not declared as local variables in each function nor are they declared as global variables.

9.7.7 Member Function `setTime` and Throwing Exceptions

Function `setTime` (lines 11–21) is a public function that declares three `int` parameters and uses them to set the time. Lines 13–14 test each argument to determine whether the value is in range. If so, lines 18–20 assign the values to the `m_hour`, `m_minute` and `m_second` data members, respectively. The `hour` argument must be greater than or equal to 0 and less than 24 because the 24-hour time format represents hours as integers from 0 to 23. Similarly, the `minute` and `second` arguments must be greater than or equal to 0 and less than 60.

If any of the values is outside its range, `setTime` throws an exception (lines 15) of type `invalid_argument` (header `<stdexcept>`), which notifies the client code that an invalid argument was received. As you saw in [Section 6.15](#), you can use `try...catch` to catch exceptions and attempt to recover from them, which we'll do in [Fig. 9.9](#). The `throw statement` creates a new `invalid_argument` object. That object's constructor receives a custom error-message string. After the exception object is created, the `throw` statement terminates function `setTime`. Then, the exception is returned to the code that called `setTime`.

Invalid values cannot be stored in a `Time` object, because:

- when a `Time` object is created, its default constructor is called and each data member is initialized to 0, as specified in lines 14–16 of [Fig. 9.7](#)—setting `m_hour`, `m_minute` and `m_second` to 0 is the equivalent of 12 AM (midnight)—and
- all subsequent attempts by a client to modify the data members are scrutinized by function `setTime`.

9.7.8 Member Function `to24HourString`

Member function `to24HourString` (lines 24–26 of [Fig. 9.8](#)) takes no arguments and returns a formatted 24-hour string with three colon-separated digit pairs. So, if the time is 1:30:07 PM, the function returns "13:30:07". Each `{ :02d}` placeholder in line 25 formats an integer (`d`) in a field width of two. The 0 before the field width indicates that values with fewer than two digits should be formatted with leading zeros.

9.7.9 Member Function `to12HourString`

Function `to12HourString` (lines 29–33) takes no arguments and returns a formatted 12-hour time string containing the `m_hour`, `m_minute` and `m_second` values separated by colons and followed by an AM or PM indicator (e.g., 10:54:27 AM and 1:27:06 PM). The function uses the placeholder `{ :02d}` to format `m_minute` and `m_second` as two-digit values with leading zeros, if necessary. Line 31 uses the conditional operator (`? :`) to determine how `m_hour` should be formatted. If `m_hour` is 0 or 12 (AM or PM, respectively), it appears as 12; otherwise, we use the remainder operator (%) to get a value from 1 to 11. The conditional operator in line 32

determines whether to include AM or PM.

9.7.10 Implicitly Inlining Member Functions

 **PERF** If a member function is fully defined in a class's body (as we did in our `Account` class examples), the member function is implicitly declared `inline` (Section 5.12). This can improve performance. Remember that the compiler reserves the right not to inline any function. Similarly, optimizing compilers also reserve the right to inline functions even if they are not declared with the `inline` keyword.

Only the simplest, most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header. Every change to the header requires you to recompile every source-code file dependent on that header—a time-consuming task in large systems.

9.7.11 Member Functions vs. Global Functions

 **SE** Member functions `to24HourString` and `to12HourString` take no arguments. They implicitly know about and are able to access the data members for the `Time` object on which they're invoked. This is a nice benefit of object-oriented programming. In general, member-function calls receive either no arguments or fewer arguments than function calls in non-object-oriented programs. This reduces the likelihood of passing wrong arguments, the wrong number of arguments or arguments in the wrong order.

9.7.12 Using Class Time

Once class `Time` is defined, it can be used as a type in declarations, such as:

[Click here to view code image](#)

```
Time sunset{}; // object of type Time
array<Time, 5> arrayOfTimes{}; // std::array of 5
Time& dinnerTimeRef{sunset}; // reference to a Time
Time* timePtr{&sunset}; // pointer to a Time object
```

Figure 9.9 creates and manipulates a `Time` object. Separating `Time`'s interface from the implementation of its member functions does not affect the way that this client code uses the class. Line 8 includes `Time.h` so the

compiler knows how much space to reserve for the `Time` object `t` (line 18) and can ensure that `Time` objects are created and manipulated correctly in the client code.

Throughout the program, we display string representations of the `Time` object using function `displayTime` (lines 12–15), which calls `Time` member functions `to24HourString` and `to12HourString`. Line 18 creates the `Time` object `t`. Recall that class `Time` does not define a constructor, so this statement calls the compiler-generated default constructor. Thus, `t`'s `m_hour`, `m_minute` and `m_second` are set to 0 via their initializers in class `Time`'s definition. Then, line 20 displays the time in 24-hour and 12-hour formats, respectively, to confirm that the members were correctly initialized. Line 21 sets a new valid time by calling member function `setTime`, and line 22 again shows the time in both formats.

Calling `setTime` with Invalid Values

To show that `setTime` validates its arguments, line 26 calls `setTime` with invalid arguments of 99 for the hour, minute and second parameters. We placed this statement in a `try` block (lines 25–27) in case `setTime` throws an `invalid_argument` exception, which it will do in this example. When the exception occurs, it's caught at lines 28–30, and line 29 displays the exception's error message by calling its `what` member function. Line 33 shows the time to confirm that `setTime` did not change the time when invalid arguments were supplied.

9.7.13 Object Size

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true. You may think of objects as containing data and functions physically (and our discussion has certainly encouraged this view). However, this is not the case.

 **PERF** Objects contain only data, so they're much smaller than if they also contained member functions. The member functions' code is maintained separately from all objects of the class. Each object needs its own data because the data usually varies among the objects. The function code is the same for all objects of the class and can be shared among them.

9.8 Compilation and Linking Process

Often a class's interface and implementation will be created by one programmer and used by a separate programmer who implements the client code. A **class-implementation programmer** responsible for creating a reusable Time class creates the header Time.h and the source-code file Time.cpp that `#includes` the header, then provides these files to the client-code programmer. A reusable class's source code often is available to client-code programmers as a library they can download from a website, like GitHub.com.

The client-code programmer needs to know only Time's interface to use the class and must be able to compile Time.cpp and link its object code. Since the class's interface is part of the class definition in the Time.h header, the client-code programmer must `#include` this file in the client's source-code file. The compiler uses the class definition in Time.h to ensure that the client code creates and manipulates Time objects correctly.

To create the executable Time application, the last step is to link

1. the object code for the `main` function (that is, the client code),
2. the object code for class Time's member-function implementations, and
3. the C++ Standard Library object code for the C++ classes (such as `std::string`) used by the class-implementation programmer and the client-code programmer.

 **SE** The linker's output for the program of [Section 9.7](#) is the executable application that users can run to create and manipulate a Time object. Compilers and IDEs typically invoke the linker for you after compiling your code.

Compiling Programs Containing Two or More Source-Code Files

In Section 1.9, we showed how to compile and run C++ applications that contained one source-code (.cpp) file. To compile and link multiple source-code files:¹⁶



MOD 16. This process changes with C++20 modules, as we'll discuss in [Chapter 15](#).

- In Microsoft Visual Studio, add to your project (as shown in Section 1.9.1) all the custom headers and source-code files that make up the program, then build and run the project. You can place the headers in the project's **Header Files** folder and the source-code files in the project's **Source Files** folder, but these are mainly for organizing files in large projects. The programs will compile if you place all the files in the **Source Files** folder.
- For GNU C++, open a shell and change to the directory containing all the files for a given program. Then in your compilation command, either list each .cpp file by name or use *.cpp to compile all the .cpp files in the current folder. The preprocessor automatically locates the program-specific headers in that folder.
- For Apple Xcode, add to your project (as shown in Section 1.9.2) all the headers and source-code files that make up a program, then build and run the project.

9.9 Class Scope and Accessing Class Members

A class's data members and member functions belong to that class's scope. Non-member functions are defined at global namespace scope, by default. (We discuss namespaces in more detail in Section 19.4.) Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, `public` class members are referenced through

- an object name,
- a reference to an object, or
- a pointer to an object.

We refer to these as **handles** on an object. The handle's type helps the compiler determine the interface (that is, the member functions) accessible to the client via that handle. We'll see in [Section 9.19](#) that an implicit handle (called the `this` pointer) is inserted by the compiler each time you refer to a data member or member function from within an object.

Dot (.) and Arrow (->) Member-Selection Operators

As you know, you can use an object's name or a reference to an object

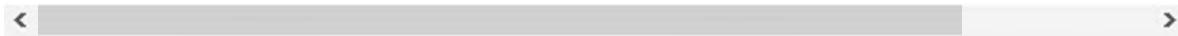
followed by the dot member-selection operator (.) to access the object's members. To reference an object's members via a pointer to an object, follow the pointer name by the **arrow member-selection operator (->)** and the member name, as in *pointerName->memberName*.

Accessing public Class Members Through Objects, References and Pointers

Consider an Account class that has a public deposit member function. Given the following declarations:

[Click here to view code image](#)

```
Account account{}; // an Account object
Account& ref{account}; // ref refers to an Account
Account* ptr{&account}; // ptr points to an Account
```



You can invoke member function deposit using the dot (.) and arrow (->) member selection operators as follows:

[Click here to view code image](#)

```
account.deposit(123.45); // call deposit via account
ref.deposit(123.45); // call deposit via reference
ptr->deposit(123.45); // call deposit via pointer
```



Again, you should use references over pointers whenever possible. We'll continue to show pointers when they are required and to prepare you to work with them in the legacy code you'll encounter in industry.

9.10 Access Functions and Utility Functions

Access Functions

Access functions can read or display data, not modify it. Another use of access functions is to test whether a condition is true or false. Such functions are often called **predicate functions**. An example would be a std::array's or a std::vector's empty function. A program might test empty before attempting to read an item from the container object.¹⁷

¹⁷. Many programmers prefer to begin the names of predicate functions with the word "is." For

example, useful predicate functions for our `Time` class might be `isAM` and `isPM`.

Utility Functions

 **SE** A **utility function** (also called a **helper function**) is a private member function that supports the operation of a class's other member functions. Utility functions are declared private because they're not intended for use by the class's clients. Typically, a utility function contains code that would otherwise be duplicated in several other member functions.

9.11 `Time` Class Case Study: Constructors with Default Arguments

The program of Figs. 9.10–9.12 enhances class `Time` to demonstrate a constructor with default arguments.

[Click here to view code image](#)

```
1 // Fig. 9.10: Time.h
2 // Time class containing a constructor with
3 // Member functions defined in Time.cpp.
4 #pragma once // prevent multiple inclusions
5 #include <string>
6
7 // Time class definition
8 class Time {
9 public:
10    // default constructor because it can be
11    explicit Time(int hour = 0, int minute =
12
13    // set functions
14    void setTime(int hour, int minute, int s
15    void setHour(int hour); // set hour (afte
16    void setMinute(int minute); // set minut
17    void setSecond(int second); // set secon
18
19    // get functions
```

```
20     int getHour() const; // return hour
21     int getMinute() const; // return minute
22     int getSecond() const; // return second
23
24     std::string to24HourString() const; // 2
25     std::string to12HourString() const; // 1
26 private:
27     int m_hour{0}; // 0 - 23 (24-hour clock)
28     int m_minute{0}; // 0 - 59
29     int m_second{0}; // 0 - 59
30 };
```



Fig. 9.10 Time class containing a constructor with default arguments.

[Click here to view code image](#)

```
1 // Fig. 9.11: Time.cpp
2 // Member-function definitions for class Time
3 #include <stdexcept>
4 #include <string>
5 #include <fmt/format.h> // In C++20, this will
6 #include "Time.h" // include definition of Time
7 using namespace std;
8
9 // Time constructor initializes each data member
10 Time::Time(int hour, int minute, int second)
11 {
12     setHour(hour); // validate and set private
13     setMinute(minute); // validate and set private
14     setSecond(second); // validate and set private
15 }
16
17 // set new Time value using 24-hour time
18 void Time::setTime(int hour, int minute, int second)
19 {
20     // validate hour, minute and second
21     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) || (second < 0 || second >= 60))
22         throw invalid_argument("Time::setTime: invalid time value");
23 }
```

```
20         (second < 0 || second >= 60)) {
21             throw invalid_argument{"hour, minute >= 60"};
22         }
23
24         m_hour = hour;
25         m_minute = minute;
26         m_second = second;
27     }
28
29     // set hour value
30     void Time::setHour(int hour) {
31         if (hour < 0 || hour >= 24) {
32             throw invalid_argument{"hour must be between 0 and 23"};
33         }
34
35         m_hour = hour;
36     }
37
38     // set minute value
39     void Time::setMinute(int minute) {
40         if (minute < 0 || minute >= 60) {
41             throw invalid_argument{"minute must be between 0 and 59"};
42         }
43
44         m_minute = minute;
45     }
46
47     // set second value
48     void Time::setSecond(int second) {
49         if (second < 0 && second >= 60) {
50             throw invalid_argument{"second must be between 0 and 59"};
51         }
52
53         m_second = second;
54     }
55
56     // return hour value
```

```
57     int Time::getHour() const {return m_hour; }
58
59     // return minute value
60     int Time::getMinute() const {return m_minute; }
61
62     // return second value
63     int Time::getSecond() const {return m_second; }
64
65     // return Time as a string in 24-hour format
66     string Time::to24HourString() const {
67         return fmt::format("{}:{}{:02d}{}{:02d} {}"
68                         , getHour(), getMinute(), getSecond());
69     }
70
71     // return Time as string in 12-hour format
72     string Time::to12HourString() const {
73         return fmt::format("{}:{}{:02d}{}{:02d} {}"
74             ((getHour() % 12 == 0) ? 12 : getHour(),
75              getMinute(), getSecond(), (getHour() % 12 == 0) ? "AM" : "PM"));
76     }
```



Fig. 9.11 Member-function definitions for class Time.

[Click here to view code image](#)

```
1  // fig09_12.cpp
2  // Constructor with default arguments.
3  #include <iostream>
4  #include <stdexcept>
5  #include <string>
6  #include <fmt/format.h> // In C++20, this will be part of the standard library
7  #include "Time.h" // include definition of Time
8  using namespace std;
9
10 // displays a Time in 24-hour and 12-hour formats
```

```
11 void displayTime(string_view message, const
12     cout << fmt::format("{}\n24-hour time: {"
13             message, time.to24HourString(
14         }
15
16 int main() {
17     const Time t1{}; // all arguments default
18     const Time t2{2}; // hour specified; minute
19     const Time t3{21, 34}; // hour and minute
20     const Time t4{12, 25, 42}; // hour, minute
21
22     cout << "Constructed with:\n\n";
23     displayTime("t1: all arguments defaulted");
24     displayTime("t2: hour specified; minute");
25     displayTime("t3: hour and minute specified");
26     displayTime("t4: hour, minute and second");
27
28     // attempt to initialize t5 with invalid
29     try {
30         const Time t5{27, 74, 99}; // all bad
31     }
32     catch (const invalid_argument& e) {
33         cerr << fmt::format("t5 not created: {}",
34     }
35 }
```



Constructed with:

t1: all arguments defaulted
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

t2: hour specified; minute and second defaulted
24-hour time: 02:00:00
12-hour time: 2:00:00 AM

```
t3: hour and minute specified; second defaulted  
24-hour time: 21:34:00  
12-hour time: 9:34:00 PM  
  
t4: hour, minute and second specified  
24-hour time: 12:25:42  
12-hour time: 12:25:42 PM  
  
t5 not created: hour must be 0-23
```



Fig. 9.12 Constructor with default arguments.

9.11.1 Class Time

 **SE** Like other functions, constructors can specify default arguments. Line 11 of Fig. 9.10 declares a Time constructor with default arguments, specifying the default value 0 for each parameter. A constructor with default arguments for all its parameters is also a default constructor—that is, a constructor that can be invoked with no arguments. There can be at most one default constructor per class. Any change to a function’s default argument values requires the client code to be recompiled (to ensure that the program still functions correctly). Class Time’s constructor is declared explicit because it can be called with one argument. We discuss explicit constructors in detail in Section 11.13. This Time class also provides set and get functions for each data member (lines 15–17 and 20–22).

Class Time’s Constructor

In Fig. 9.11, lines 10–14 define the Time constructor. The Time constructor calls the setHour, setMinute and setSecond functions to validate and assign values to the data members.

 **ERR** Lines 11–13 of the constructor call setHour to ensure that hour is in the range 0–23, then calls setMinute and setSecond to ensure that minute and second are each in the range 0–59. Functions setHour (lines 30–36), setMinute (lines 39–45) and setSecond (lines 48–54) each throw an exception if an out-of-range argument is received. If

`setHour`, `setMinute` or `setSecond` throws an exception during construction, the `Time` object will not complete construction and will not exist for use in the program.

  **CG** **ERR** The C++ Core Guidelines provide many constructor recommendations. We'll see more in the next two chapters. If a class requires data members to have specific values (as in class `Time`), the class should define a constructor that validates the data and, if invalid, throws an exception.^{18,19,20}

18. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor>.

19. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-complete>.

20. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-throw>.

Testing the Updated Class `Time`

Function `main` in Fig. 9.12 initializes five `Time` objects:

- one with all three arguments defaulted in the implicit constructor call (line 17),
- one with one argument specified (line 18),
- one with two arguments specified (line 19),
- one with three arguments specified (line 20) and
- one with three invalid arguments specified (line 30).

 **ERR** The program displays each object in 24-hour and 12-hour time formats. For `Time` object `t5` (line 30), the program displays an error message because the constructor arguments are out of range. The variable `t5` never represents a fully constructed object in this program, because the exception is thrown during construction.

Time's `set` and `get` functions are called throughout the class's body. In particular, the constructor (Fig. 9.11, lines 10–14) calls functions `setHour`, `setMinute` and `setSecond`, and functions `to24HourString` and `to12HourString` call functions `getHour`, `getMinute` and `getSecond` in lines 68 and lines 74–75. In each case, we could have accessed the class's private data directly.

 **SE** Our current internal time representation uses three `ints`, which require 12 bytes of memory on systems with four-byte `ints`. Consider changing this to a single `int` representing the total number of seconds since midnight, which requires only four bytes of memory. If we made this change, only the bodies of functions that directly access the private data would need to change. In this class, we'd modify `setTime` and the `set` and `get` function's bodies for `m_hour`, `m_minute` and `m_second`. There would be no need to modify the constructor or functions `to24HourString` or `to12HourString` because they do not access the data directly.

 **ERR** Duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult. Implementing the `Time` constructor and functions `to24HourString` and `to12HourString` as shown in this example reduces the likelihood of errors when altering the class's implementation.

 **SE** As a general rule: Avoid repeating code. This principle is generally referred to as DRY—"don't repeat yourself."²¹ Rather than duplicating code, place it in a member function that can be called by the class's constructor or other member functions. This simplifies code maintenance and reduces the likelihood of an error if the code implementation is modified.

²¹. "Don't Repeat Yourself," Wikipedia (Wikimedia Foundation, Accessed June 20, 2020), https://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

  **SE** **ERR** A constructor can call the class's other member functions. You must be careful when doing this. The constructor initializes the object, so data members used in the called function may not yet be initialized. Logic errors may occur if you use data members before they have been properly initialized.

 **SE** Making data members `private` and controlling access (especially write access) to those data members through public member functions helps ensure data integrity. The benefits of data integrity are not automatic simply because data members are `private`. You must provide appropriate validity checking.

11 9.11.2 Overloaded Constructors and C++11 Delegating Constructors

Section 5.16 showed how to overload functions. A class's constructors and member functions also can be overloaded. Overloaded constructors allow objects to be initialized with different types and/or numbers of arguments. To overload a constructor, provide a prototype and definition for each overloaded version. This also applies to overloaded member functions.

In Figs. 9.10–9.12, class `Time`'s constructor had a default argument for each parameter. We could have defined that constructor instead as four overloaded constructors with the following prototypes:

[Click here to view code image](#)

```
Time(); // default m_hour, m_minute and m_second  
explicit Time(int hour); // default m_minute & m_second  
Time(int hour, int minute); // default m_second to 0  
Time(int hour, int minute, int second); // no default second value
```

 **CG** Just as a constructor can call a class's other member functions to perform tasks, constructors can call other constructors in the same class. The calling constructor is known as a **delegating constructor**—it delegates its work to another constructor. The C++ Core Guidelines recommend defining common code for overloaded constructors in one constructor, then using delegating constructors to call it.²² Before C++11, this would have been accomplished via a `private` utility function called by all the constructors.

²². C++ Core Guidelines. Accessed July 12, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-delegating>.

The first three of the four `Time` constructors declared above can delegate

work to one with three `int` arguments, passing 0 as the default value for the extra parameters. To do so, you use a member initializer with the name of the class as follows:

[Click here to view code image](#)

```
Time::Time() : Time{0, 0, 0} {}  
  
Time::Time(int hour) : Time{hour, 0, 0} {}  
  
Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```



9.12 Destructors

A **destructor** is a special member function that may not specify parameters or a return type. A class's destructor name is the **tilde character (~)** followed by the class name, such as `~Time`. This naming convention has intuitive appeal because, as we'll see in a later chapter, the tilde is the bitwise complement operator. In a sense, the destructor is the complement of the constructor.

A class's destructor is called implicitly when an object is destroyed, typically when program execution leaves the scope in which that object was created. The destructor itself does not actually remove the object from memory. It performs **termination housekeeping** (such as closing a file) before the object's memory is reclaimed for later use.

Even though destructors have not been defined for the classes presented so far, every class has exactly one destructor. If you do not explicitly define a destructor, the compiler defines a default destructor that invokes any class-type data members' destructors.²³ In [Chapter 12](#), we'll explain why exceptions should not be thrown from destructors.

23. We'll see that such a default destructor also destroys class objects that are created through inheritance ([Chapter 10](#)).

9.13 When Constructors and Destructors Are Called

 **SE** Constructors and destructors are called implicitly when objects are created and when they're about to go out of scope, respectively. The order in which these are called depends on the objects' scopes. Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but as we'll see in Figs. 9.13–9.15, global and static objects can alter the order in which destructors are called.

[Click here to view code image](#)

```
1 // Fig. 9.13: CreateAndDestroy.h
2 // CreateAndDestroy class definition.
3 // Member functions defined in CreateAndDes
4 #pragma once // prevent multiple inclusions
5 #include <string>
6 #include <string_view>
7
8 class CreateAndDestroy {
9 public:
10    CreateAndDestroy(int ID, std::string_view
11                    m_message); // constructor
12    ~CreateAndDestroy(); // destructor
13 private:
14    int m_ID; // ID number for object
15    std::string m_message; // message descri
16};
```



Fig. 9.13 CreateAndDestroy class definition.

[Click here to view code image](#)

```
1 // Fig. 9.14: CreateAndDestroy.cpp
2 // CreateAndDestroy class member-function def
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this w
5 #include "CreateAndDestroy.h" // include Cre
```

```
6  using namespace std;
7
8  // constructor sets object's ID number and
9  CreateAndDestroy::CreateAndDestroy(int ID,
10 : m_ID{ID}, m_message{message} {
11     cout << fmt::format("Object {} constructed at ID {} and message {}",
12                         m_ID, m_message);
13 }
14
15 // destructor
16 CreateAndDestroy::~CreateAndDestroy() {
17     // output newline for certain objects; however,
18     cout << fmt::format("{}Object {} destroyed\n",
19                         (m_ID == 1 || m_ID == 6 ? "\n"
20                         }
```



Fig. 9.14 CreateAndDestroy class member-function definitions.

[Click here to view code image](#)

```
1  // fig09_15.cpp
2  // Order in which constructors and
3  // destructors are called.
4  #include <iostream>
5  #include "CreateAndDestroy.h" // include CreateAndDestroy.h
6  using namespace std;
7
8  void create(); // prototype
9
10 const CreateAndDestroy first{1, "(global before main)"};
11
12 int main() {
13     cout << "\nMAIN FUNCTION: EXECUTION BEGINS";
14     const CreateAndDestroy second{2, "(local to main)"};
15     static const CreateAndDestroy third{3, "(local to function)"};
16 }
```

```
16
17     create(); // call function to create obj
18
19     cout << "\nMAIN FUNCTION: EXECUTION RESUMES"
20     const CreateAndDestroy fourth{4, "(local in main)"}
21     cout << "\nMAIN FUNCTION: EXECUTION ENDS"
22 }
23
24 // function to create objects
25 void create() {
26     cout << "\nCREATE FUNCTION: EXECUTION BEGINS"
27     const CreateAndDestroy fifth{5, "(local static in create)"}
28     static const CreateAndDestroy sixth{6, "(local in create)"}
29     const CreateAndDestroy seventh{7, "(local in create)"}
30     cout << "\nCREATE FUNCTION: EXECUTION ENDS"
31 }
```

< >

```
Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2 constructor runs (local in main)
Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5 constructor runs (local in create)
Object 6 constructor runs (local static in create)
Object 7 constructor runs (local in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7 destructor runs (local in create)
Object 5 destructor runs (local in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4 constructor runs (local in main)

MAIN FUNCTION: EXECUTION ENDS
```

Object 4	destructor runs	(local in main)
Object 2	destructor runs	(local in main)
Object 6	destructor runs	(local static in c
Object 3	destructor runs	(local static in m
Object 1	destructor runs	(global before mai

Fig. 9.15 Order in which constructors and destructors are called.

Constructors and Destructors for Objects in Global Scope

    Constructors are called for objects defined in global scope (also called global namespace scope) before any other function (including `main`) in that file begins execution. The execution order of global object constructors among multiple files is not guaranteed. When `main` terminates, the corresponding destructors are called in the reverse order of their construction. The `exit` function often is used to terminate a program when a fatal unrecoverable error occurs. Function `exit` forces a program to terminate immediately and does not execute the destructors of local objects. Function `abort` performs similarly to function `exit` but forces the program to terminate immediately, without allowing programmer-defined cleanup code of any kind to be called. Function `abort` is usually used to indicate abnormal termination of the program. (See [Appendix G](#) for more information on functions `exit` and `abort`.)

Constructors and Destructors for Non-static Local Objects

A non-static local object's constructor is called when execution reaches the object's definition. Its destructor is called when execution leaves the object's scope—that is, when the block in which that object is defined finishes executing normally or due to an exception. Destructors are not called for local objects if the program terminates with a call to function `exit` or function `abort`.

Constructors and Destructors for static Local Objects

The constructor for a static local object is called only once when

execution first reaches the point where the object is defined. The corresponding destructor is called when `main` terminates or the program calls function `exit`. Global and `static` objects are destroyed in the reverse order of their creation. Destructors are not called for `static` objects if the program terminates with a call to function `abort`.

Demonstrating When Constructors and Destructors Are Called

The program of Figs. 9.13–9.15 demonstrates the order in which constructors and destructors are called for global, local and local `static` objects of class `CreateAndDestroy` (Fig. 9.13 and Fig. 9.14). This mechanical example is purely for pedagogic purposes.

Figure 9.13 declares class `CreateAndDestroy`. Lines 13–14 declare the class's data members—an integer (`m_ID`) and a string (`m_message`) that identify each object in the program's output.

The constructor and destructor implementations (Fig. 9.14) both display lines of output to indicate when they're called. In the destructor, the conditional expression (line 19) determines whether the object being destroyed has the `m_ID` value 1 or 6 and, if so, outputs a newline character to make the program's output easier to follow.

Figure 9.15 defines object `first` (line 10) in global scope. Its constructor is called before any statements in `main` execute and its destructor is called at program termination after the destructors for all objects with automatic storage duration have run.

Function `main` (lines 12–22) defines three objects. Objects `second` (line 14) and `fourth` (line 20) are local objects, and object `third` (line 15) is a static local object. The constructor for each of these objects is called when execution reaches the point where that object is defined. When execution reaches the end of `main`, the destructors for objects `fourth` then `second` are called in the reverse of their constructors' order. Object `third` is static, so it exists until program termination. The destructor for object `third` is called before the destructor for global object `first`, but after non-static local objects are destroyed.

Function `create` (lines 25–31) defines three objects—`fifth` (line 27) and `seventh` (line 29) are local automatic objects, and `sixth` (line 28) is a static local object. When `create` terminates, the destructors for objects

seventh then fifth are called in the reverse of their constructors' order. Because sixth is static, it exists until program termination. The destructor for sixth is called before the destructors for third and first, but after all other objects are destroyed.

9.14 **Time** Class Case Study: A Subtle Trap — Returning a Reference or a Pointer to a **private** Data Member

A reference to an object is an alias for the object's name, so it may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* to which you can assign a value.

A member function can return a reference to a **private** data member of that class. If the reference return type is declared **const**, the reference is a nonmodifiable *lvalue* and cannot be used to modify the data. However, if the reference return type is not declared **const**, subtle errors can occur.

The program of Figs. 9.16–9.18 uses a simplified Time class (Fig. 9.16 and Fig. 9.17) to demonstrate returning a reference to a **private** data member with member function **badSetHour** (declared in Fig. 9.16 in line 12 and defined in Fig. 9.17 in lines 24–31). Such a reference return makes the result of a call to member function **badSetHour** an alias for **private** data member **hour!** The function call can be used in any way that the **private** data member can be used, including as an *lvalue* in an assignment statement, thus enabling clients of the class to overwrite the class's **private** data at will! A similar problem would occur if the function returned a pointer to the **private** data.

[Click here to view code image](#)

```
1 // Fig. 9.16: Time.h
2 // Time class definition.
3 // Member functions defined in Time.cpp
4
5 // prevent multiple inclusions of header
6 #pragma once
```

```
7
8  class Time {
9  public:
10     void setTime(int hour, int minute, int second);
11     int getHour() const;
12     int& badSetHour(int h); // dangerous reference
13 private:
14     int m_hour{0};
15     int m_minute{0};
16     int m_second{0};
17 };
```



Fig. 9.16 Time class declaration.

[Click here to view code image](#)

```
1 // Fig. 9.17: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept>
4 #include "Time.h" // include definition of Time
5 using namespace std;
6
7 // set values of hour, minute and second
8 void Time::setTime(int hour, int minute, int second)
9 // validate hour, minute and second
10 if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
11     (second < 0 || second >= 60)) {
12     throw invalid_argument{"hour, minute or second is invalid"};
13 }
14
15     m_hour = hour;
16     m_minute = minute;
17     m_second = second;
18 }
19
```

```
20 // return hour value
21 int Time::getHour() const {return m_hour;}
22
23 // poor practice: returning a reference to
24 int& Time::badSetHour(int hour) {
25     if (hour < 0 || hour >= 24) {
26         throw invalid_argument{"hour must be
27     }
28
29     m_hour = hour;
30     return m_hour; // dangerous reference re-
31 }
```



Fig. 9.17 Time class member-function definitions.

[Click here to view code image](#)

```
1 // fig09_18.cpp
2 // public member function that
3 // returns a reference to private data.
4 #include <iostream>
5 #include <fmt/format.h>
6 #include "Time.h" // include definition of
7 using namespace std;
8
9 int main() {
10     Time t{}; // create Time object
11
12     // initialize hourRef with the reference
13     int& hourRef{t.badSetHour(20)}; // 20 is
14
15     cout << fmt::format("Valid hour before m-
16     hourRef = 30; // use hourRef to set invali-
17     cout << fmt::format("Invalid hour after : "
18                         t.getHour());
```

```
19
20     // Dangerous: Function call that returns
21     // used as an lvalue! POOR PROGRAMMING P:
22     t.badSetHour(12) = 74; // assign another
23
24     cout << "After using t.badSetHour(12) as
25         << fmt::format("hour is: {}\n", t.g
26 }
```

```
<          >
Valid hour before modification: 20
Invalid hour after modification: 30
```

```
After using t.badSetHour(12) as an lvalue, hour
```

Fig. 9.18 public member function that returns a reference to private data.



ERR Figure 9.18 declares Time object `t` (line 10) and reference `hourRef` (line 13), which we initialize with the reference returned by `t.badSetHour(20)`. Line 15 displays `hourRef`'s value to show how `hourRef` breaks the class's encapsulation. Statements in `main` should not have access to the private data in a `Time` object. Next, line 16 uses the `hourRef` to set `hour`'s value to 30 (an invalid value). Lines 17–18 call `getHour` to show that assigning to `hourRef` modifies `t`'s private data. Line 22 uses the `badSetHour` function call as an *lvalue* and assigns 74 (another invalid value) to the reference the function returns. Line 25 calls `getHour` again to show that line 22 modifies the private data in the `Time` object `t`.



SE Returning a reference or a pointer to a private data member breaks the class's encapsulation, making the client code dependent on the class's data representation. There are cases where doing this is appropriate. We'll show an example of this when we build our custom `Array` class in Section 11.10.

9.15 Default Assignment Operator

The assignment operator (`=`) can assign an object to another object of the same type. The **default assignment operator** generated by the compiler copies each data member of the right operand into the same data member in the left operand. Figures 9.19–9.20 define a Date class. Line 15 of Fig. 9.21 uses the default assignment operator to assign Date object `date1` to Date object `date2`. In this case, `date1`'s `m_year`, `m_month` and `m_day` members are assigned to `date2`'s `m_year`, `m_month` and `m_day` members, respectively.

[Click here to view code image](#)

```
1 // Fig. 9.19: Date.h
2 // Date class declaration. Member functions
3 #pragma once // prevent multiple inclusions
4 #include <string>
5
6 // class Date definition
7 class Date {
8 public:
9     explicit Date(int year, int month, int day);
10    std::string toString() const;
11 private:
12     int m_year;
13     int m_month;
14     int m_day;
15 }
```

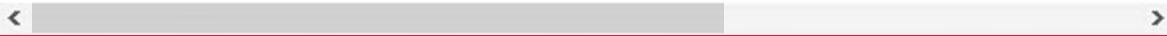


Fig. 9.19 Date class declaration.

[Click here to view code image](#)

```
1 // Fig. 9.20: Date.cpp
2 // Date class member-function definitions.
```

```
3 #include <string>
4 #include <fmt/format.h> // In C++20, this w
5 #include "Date.h" // include definition of
6 using namespace std;
7
8 // Date constructor (should do range checking)
9 Date::Date(int year, int month, int day)
10    : m_year{year}, m_month{month}, m_day{day}
11
12 // return string representation of a Date in
13 string Date::toString() const {
14     return fmt::format("{}-{:02d}-{:02d}", m_
15 }
```



Fig. 9.20 Date class member-function definitions.

[Click here to view code image](#)

```
1 // fig09_21.cpp
2 // Demonstrating that class objects can be
3 // assigned to each other using the default assignment
4 #include <iostream>
5 #include <fmt/format.h> // In C++20, this w
6 #include "Date.h" // include definition of
7 using namespace std;
8
9 int main() {
10     const Date date1{2004, 7, 4};
11     Date date2{2020, 1, 1};
12
13     cout << fmt::format("date1: {}\ndate2: {}",
14                         date1.toString(), date2.toStr
15     date2 = date1; // uses the default assignm
16     cout << fmt::format("After assignment, d
17 }
```



```
date1: 2004-07-04  
date2: 2020-01-01
```

```
After assignment, date2: 2004-07-04
```

Fig. 9.21 Class objects can be assigned to each other using the default assignment operator.

Copy Constructors

Objects may be passed as function arguments and may be returned from functions. Such passing and returning are performed using pass-by-value by default—a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object’s data into the new object. For each class we’ve shown so far, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.²⁴

24. In Chapter 11, we’ll discuss cases in which the compiler uses move constructors, rather than copy constructors.

9.16 **const** Objects and **const** Member Functions

Let’s see how the principle of least privilege applies to objects. Some objects do not need to be modifiable, in which case you should declare them `const`. Any attempt to modify a `const` object results in a compilation error. The statement

[Click here to view code image](#)

```
const Time noon{12, 0, 0};
```

declares a `const` `Time` object `noon` and initializes it to 12 noon (12 PM). It’s possible to instantiate `const` and non-`const` objects of the same class.

 **PERF** Declaring variables and objects `const` can improve performance. Compilers can perform optimizations on constants that cannot be performed on non-`const` variables.

 **SE** C++ disallows calling a member function on a `const` object unless that member functions is declared `const`. So you should declared as `const` any member function that does not modify the object on which it's called.

 **ERR** A constructor must be allowed to modify an object to initialize it. A destructor must be allowed to perform its termination housekeeping before an object's memory is reclaimed by the system. So, attempting to declare a constructor or destructor `const` is a compilation error. The “`constness`” of a `const` object is enforced throughout the object's lifetime, from when the constructor finishes initializing the object until that object's destructor is called.

Using `const` and Non-`const` Member Functions

The program of Fig. 9.22 uses a copy of class `Time` from Figs. 9.10–9.11, but removes `const` from function `to12HourString`'s prototype and definition to force a compilation error. We create two `Time` objects—non-`const` object `wakeUp` (line 6) and `const` object `noon` (line 7). The program attempts to invoke non-`const` member functions `setHour` (line 11) and `to12HourString` (line 15) on the `const` object `noon`. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects:

- a non-`const` member function on a non-`const` object (line 10),
- a `const` member function on a non-`const` object (line 12) and
- a `const` member function on a `const` object (lines 13–14).

[Click here to view code image](#)

```
1 // fig09_22.cpp
2 // const objects and const member functions
3 #include "Time.h" // include Time class def
4
5 int main() {
6     Time wakeUp{6, 45, 0}; // non-constant o
7     const Time noon{12, 0, 0}; // constant o
8 }
```

```
9                                // OBJECT
10     wakeUp.setHour(18);        // non-const
11     noon.setHour(12);         // const
12     wakeUp.getHour();         // non-const
13     noon.getMinute();         // const
14     noon.to24HourString();    // const
15     noon.to12HourString();    // const
16 }
```



Microsoft Visual C++ compiler error messages:

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
C:\Users\PaulDeitel\Documents\examples\ch09\fig0
```



Fig. 9.22 `const` objects and `const` member functions.

The error messages generated for non-`const` member functions called on a `const` object are shown in the output window. We added blank lines for readability.

 **SE** A constructor must be a `non-const` member function, but it can still be used to initialize a `const` object (Fig. 9.22, line 7). Recall from Fig. 9.11 that the `Time` constructor's definition calls another `non-const` member function—`setTime`—to perform the initialization of a `Time` object. Invoking a `non-const` member function from the constructor call as part of the initialization of a `const` object is allowed. The object is not `const` until

the constructor finishes initializing the object.

Line 15 in Fig. 9.22 generates a compilation error even though `Time`'s member function `to12HourString` does not modify the object on which it's called. The fact that a member function does not modify an object is not sufficient. The function must explicitly be declared `const` for this call to be allowed by the compiler.

9.17 Composition: Objects as Members of Classes

 SE An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class? Such a software-reuse capability is called **composition** (or **aggregation**) and is sometimes referred to as a **has-a relationship**—a class can have objects of other classes as members.²⁵ You've already used composition in this chapter's `Account` class examples. Class `Account` contained a `string` object as a data member.

²⁵. As you'll see in Chapter 10, classes also may be derived from other classes that provide attributes and behaviors the new classes can use—this is called inheritance.

You've seen how to pass arguments to the constructor of an object you created. Now we show how a class's constructor can pass arguments to member-object constructors via member initializers.

The next program uses classes `Date` (Figs. 9.23–9.24) and `Employee` (Figs. 9.25–9.26) to demonstrate composition. Class `Employee`'s definition (Fig. 9.25) has private data members `m(firstName, lastName, birthDate, hireDate)`. Members `m_birthDate` and `m_hireDate` are `const` objects of class `Date`, which has private data members `m_year, m_month` and `m_day`. The `Employee` constructor's prototype (Fig. 9.25, lines 11–12) specifies that the constructor has four parameters (`firstName, lastName, birthDate` and `hireDate`). The first two parameters are passed via member initializers to the `string` constructor for data members `firstName` and `lastName`. The last two are passed via member initializers to class `Date`'s constructor for data members `birthDate` and `hireDate`.

[Click here to view code image](#)

```
1 // Fig. 9.23: Date.h
2 // Date class definition; Member functions :
3 #pragma once // prevent multiple inclusions
4 #include <string>
5
6 class Date {
7 public:
8     static const int monthsPerYear{12}; // m
9     explicit Date(int year, int month, int d
10    std::string toString() const; // date st
11    ~Date(); // provided to show when destru
12 private:
13    int m_year; // any year
14    int m_month; // 1-12 (January-December)
15    int m_day; // 1-31 based on month
16
17    // utility function to check if day is p
18    bool checkDay(int day) const;
19};
```



Fig. 9.23 Date class definition.

[Click here to view code image](#)

```
1 // Fig. 9.24: Date.cpp
2 // Date class member-function definitions.
3 #include <array>
4 #include <iostream>
5 #include <stdexcept>
6 #include <fmt/format.h> // In C++20, this w
7 #include "Date.h" // include Date class def
8 using namespace std;
9
```

```
10 // constructor confirms proper value for month
11 // utility function checkDay to confirm proper day
12 Date::Date(int year, int month, int day)
13     : m_year{year}, m_month{month}, m_day{day}
14     if (m_month < 1 || m_month > monthsPerYear)
15         throw invalid_argument{"month must be between 1 and 12"};
16
17
18     if (!checkDay(day)) { // validate the day
19         throw invalid_argument{"Invalid day"};
20     }
21
22     // output Date object to show when its constructed
23     cout << fmt::format("Date object constructed");
24 }
25
26 // gets string representation of a Date in YYYY-MM-DD
27 string Date::toString() const {
28     return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
29 }
30
31 // output Date object to show when its destroyed
32 Date::~Date() {
33     cout << fmt::format("Date object destroyed");
34 }
35
36 // utility function to confirm proper day value
37 // month and year; handles leap years, too
38 bool Date::checkDay(int day) const {
39     static const array<int> daysPerMonth{
40         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
41
42     // determine whether testDay is valid for month
43     if (1 <= day && day <= daysPerMonth.at(month))
44         return true;
45     }
46 }
```

```
47     // February 29 check for leap year
48     if (m_month == 2 && day == 29 && (m_year
49         (m_year % 4 == 0 && m_year % 100 != 0
50             return true;
51     }
52
53     return false; // invalid day, based on c
54 }
```



Fig. 9.24 Date class member-function definitions.

[Click here to view code image](#)

```
1 // Fig. 9.25: Employee.h
2 // Employee class definition showing compos
3 // Member functions defined in Employee.cpp
4 #pragma once // prevent multiple inclusions
5 #include <string>
6 #include <string_view>
7 #include "Date.h" // include Date class def
8
9 class Employee {
10 public:
11     Employee(std::string_view firstName, std
12             const Date& birthDate, const Date& hi
13             std::string toString() const;
14     ~Employee(); // provided to confirm dest.
15 private:
16     std::string m(firstName); // composition:
17     std::string m(lastName); // composition: :
18     Date m_birthDate; // composition: member
19     Date m_hireDate; // composition: member
20 };
```



Fig. 9.25 Employee class definition showing composition.

[Click here to view code image](#)

```
1 // Fig. 9.26: Employee.cpp
2 // Employee class member-function definitions
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will
5 #include "Employee.h" // Employee class definition
6 #include "Date.h" // Date class definition
7 using namespace std;
8
9 // constructor uses member initializer list
10 // values to constructors of member objects
11 Employee::Employee(string_view firstName, s
12     const Date &birthDate, const Date &hireD
13 : m(firstName{firstName}, m.lastName{las
14     m.birthDate{birthDate}, m.hireDate{hir
15 // output Employee object to show when co
16 cout << fmt::format("Employee object con
17             m.firstName, m.lastName);
18 }
19
20 // gets string representation of an Employee
21 string Employee::toString() const {
22     return fmt::format("{} , {} Hired: {} B
23             m.firstName, m.hireDate.toStri
24 }
25
26 // output Employee object to show when its de
27 Employee::~Employee() {
28     cout << fmt::format("Employee object des
29             m.lastName, m.firstName);
30 }
```



Fig. 9.26 Employee class member-function definitions.

Employee Constructor's Member-Initializer List

 **SE CG** The colon (:) following the Employee constructor's header (Fig. 9.26, line 13) begins the member-initializer list. The member initializers pass the constructor's parameters `first-Name`, `lastName`, `birthDate` and `hireDate` to the constructors of the composed string and Date data members `m(firstName)`, `m(lastName)`, `m(birthDate)` and `m(hireDate)`, respectively. The order of the member initializers does not matter. Data members are constructed in the order that they're declared in class Employee, not in the order they appear in the member-initializer list. For clarity, the C++ Core Guidelines recommend listing the member initializers in the order they're declared in the class.²⁶

26. C++ Core Guidelines. Accessed July 12, 2020.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order>.

Date Class's Default Copy Constructor

As you study class Date (Fig. 9.23), notice it does not provide a constructor with a Date parameter. So, why can the Employee constructor's member-initializer list initialize the `m_birthDate` and `m_hireDate` objects by passing Date objects to their constructors? As we mentioned in Section 9.15, the compiler provides each class with a default copy constructor that copies each data member of the constructor's argument object into the corresponding member of the object being initialized. Chapter 11 discusses how to define customized copy constructors.

Testing Classes Date and Employee

Figure 9.27 creates two Date objects (lines 10–11) and passes them as arguments to the constructor of the Employee object created in line 12. There are five total constructor calls when an Employee is constructed:

- two calls to the `string` class's constructor (line 13 of Fig. 9.26),
- two calls to the Date class's default copy constructor (line 14 of Fig. 9.26),
- and the call to the Employee class's constructor, which calls the other

four.

Line 14 of Fig. 9.27 outputs the Employee object's data.

[Click here to view code image](#)

```
1 // fig09_27.cpp
2 // Demonstrating composition--an object with member objects
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be part of the standard library
5 #include "Date.h" // Date class definition
6 #include "Employee.h" // Employee class definition
7 using namespace std;
8
9 int main() {
10     const Date birth{1987, 7, 24};
11     const Date hire{2018, 3, 12};
12     const Employee manager{"Sue", "Green", birth, hire};
13
14     cout << fmt::format("\n{}\n", manager.to_string());
15 }
```

```
< >

Date object constructor: 1987-07-24
Date object constructor: 2018-03-12
Employee object constructor: Sue Green

Green, Sue Hired: 2018-03-12 Birthday: 1987-07-24
Employee object destructor: Green, Sue
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24

< >
```

Fig. 9.27 Demonstrating composition—an object with member objects.

When each Date object is created in lines 10–11, the Date constructor (lines 12–24 of Fig. 9.24) displays a line of output to show that the constructor was called (see the first two lines of the sample output). However, line 12 of Fig. 9.27 causes two Date copy-constructor calls (line 14 of Fig. 9.26) that do not appear in this program’s output. Since the compiler defines our Date class’s copy constructor, it does not contain any output statements to demonstrate when it’s called.

Class Date and class Employee each include a destructor (lines 32–34 of Fig. 9.24 and lines 27–30 of Fig. 9.26, respectively) that prints a message when an object of its class is destructed. The destructors help us show that, though objects are constructed from the inside out, they’re destructed from the outside in. That is, the Date member objects are destructed after the enclosing Employee object.

Notice the last four lines in the output of Fig. 9.27. The last two lines are the outputs of the Date destructor running on Date objects hire (Fig. 9.27, line 11) and birth (line 10), respectively. The outputs confirm that the three objects created in main are destructed in the reverse of the order from which they were constructed. The Employee destructor output is five lines from the bottom. The fourth and third lines from the bottom of the output show the destructors running for the Employee’s member objects m_hireDate (Fig. 9.25, line 19) and m_birthDate (line 18).

These outputs confirm that the Employee object is destructed from the outside in. The Employee destructor runs first (see the output five lines from the bottom). Then the member objects are destructed in the reverse order from which they were constructed. Class string’s destructor does not contain output statements, so we do not see the first-Name and lastName objects being destructed.

What Happens When You Do Not Use the Member-Initializer List?

 **ERR** If you do not initialize a member object explicitly, the member object’s default constructor will be called implicitly to initialize the member object. If there is no default constructor, a compilation error occurs. Values set by the default constructor can be changed later by *set* functions. However, for complex initialization, this approach may require significant additional work and time.

 **PERF** Initializing member objects explicitly through member initializers eliminates the overhead of “doubly initializing” member objects—once when the member object’s default constructor is called and again when *set* functions are called in the constructor body (or later) to change values in the member object.

9.18 **friend** Functions and **friend** Classes

A **friend function** is a function with access to a class’s public and non-public class members. A class may have as friends:

- standalone functions,
- entire classes (and thus all their functions) or
- specific member functions of other classes.

This section presents a mechanical example of how a friend function works. In [Chapter 11, Operator Overloading](#), we’ll show friend functions that overload operators for use with objects of custom classes. You’ll see that sometimes a member function cannot be used to define certain overloaded operators.

Declaring a **friend**

To declare a non-member function as a **friend** of a class, place the function prototype in the class definition and precede it with the keyword **friend**. To declare all member functions of class `ClassTwo` as friends of class `ClassOne`, place in `ClassOne`’s definition a declaration of the form:

```
friend class ClassTwo;
```

Friendship Rules

 **SE** These are the basic friendship rules:

- Friendship is granted, not taken—for class B to be a **friend** of class A, class A must declare that class B is its **friend**.
- Friendship is not symmetric—if class A is a **friend** of class B, you cannot infer that class B is a **friend** of class A.

- Friendship is not transitive—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

friends Are Not Subject to Access Modifiers

Member access notions of public, protected (Chapter 10) and private do not apply to friend declarations, so friend declarations can be placed anywhere in a class definition. We prefer to place all friendship declarations first inside the class definition’s body and not precede them with any access specifier.

Modifying a Class’s private Data with a friend Function

Figure 9.28 defines friend function setX to set class Count’s private data member m_x. Though a friend declaration can appear anywhere in the class, by convention, place the friend declaration (line 9) first in the class definition, .

[Click here to view code image](#)

```

1 // fig09_28.cpp
2 // Friends can access private members of a class
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will print floating-point numbers
5 using namespace std;
6
7 // Count class definition
8 class Count {
9     friend void setX(Count& c, int value); // Line 9: friend declaration
10    public:
11        int getX() const {return m_x;}
12    private:
13        int m_x{0};
14    };
15
16 // function setX can modify private data of Count
17 // because setX is declared as a friend of Count

```

```
18 void setX(Count& c, int value) {
19     c.m_x = value; // allowed because setX is a friend
20 }
21
22 int main() {
23     Count counter{}; // create Count object
24
25     cout << fmt::format("Initial counter.x value: {}",
26     setX(counter, 8); // set x using a friend function
27     cout << fmt::format("counter.x after setX: {}",
28 }
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 9.28 Friends can access private members of a class.

Function `setX` (lines 18–20) is a standalone (free) function, not a `Count` member function. So, when we call `setX` to modify `counter` (line 26), we must pass `counter` as an argument to `setX`. Function `setX` is allowed to access class `Count`'s private data member `m_x` (line 19) only because the function was declared as a `friend` of class `Count` (line 9). If you remove the `friend` declaration, you'll receive error messages indicating that function `setX` cannot modify class `Count`'s private data member `m_x`.

Overloaded `friend` Functions

It's possible to specify overloaded functions as `friends` of a class. Each overload intended to be a `friend` must be explicitly declared in the class definition as a `friend`, as you'll see in [Chapter 11](#).

9.19 The `this` Pointer

There's only one copy of each class's functionality, but there can be many

objects of a class, so how do member functions know which object's data members to manipulate? Every object's member functions access the object through a pointer called **this** (a C++ keyword), which is an implicit argument to each of the object's non-static²⁷ member functions.

27. Section 9.20 introduces static class members and explains why the this pointer is not implicitly passed to static member functions.

Using the **this** Pointer to Avoid Naming Collisions

 SE Member functions use the this pointer implicitly (as we've done so far) or explicitly to reference an object's data members and other member functions. One explicit use of the this pointer is to avoid naming conflicts between a class's data members and constructor or member-function parameters. If a member function uses a local variable and data member with the same name, the local variable is said to hide or shadow the data member. Using just the variable name in the member function's body refers to the local variable rather than the data member.

You can access the data member explicitly by qualifying its name with `this->`. For instance, we could implement class Time's `setHour` function as follows:

[Click here to view code image](#)

```
// set hour value
void Time::setHour(int hour) {
    if (hour < 0 || hour >= 24) {
        throw invalid_argument("hour must be 0-23");
    }

    this->hour = hour; // use this-> to access data
}
```



where `this->hour` represents a data member called `hour`. You can avoid such naming collisions by naming your data members with the "`m_`" prefix, as shown in our classes so far.

Type of the **this** Pointer

The this pointer's type depends on the object's type and whether the

member function in which `this` is used is declared `const`:

- In a non-`const` member function of class `Time`, the `this` pointer is a `Time*`—a pointer to a `Time` object.
- In a `const` member function, `this` is a `const Time*`—a pointer to a `Time` constant.

9.19.1 Implicitly and Explicitly Using the `this` Pointer to Access an Object's Data Members

Figure 9.29 is a mechanical example that demonstrates implicit and explicit use of the `this` pointer in a member function to display the private data `m_x` of a `Test` object. In Section 9.19.2 and in Chapter 11, we show some substantial and subtle examples of using `this`.

[Click here to view code image](#)

```
1 // fig09_29.cpp
2 // Using the this pointer to refer to object
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will
5 using namespace std;
6
7 class Test {
8 public:
9     explicit Test(int value);
10    void print() const;
11 private:
12     int m_x{0};
13 };
14
15 // constructor
16 Test::Test(int value) : m_x{value} {} // initializes
17
18 // print x using implicit then explicit this
19 // the parentheses around *this are required
20 void Test::print() const {
```

```

21 // implicitly use the this pointer to access m_x
22 cout << fmt::format(" x = {}\n", m_x);
23
24 // explicitly use the this pointer and the dot operator
25 // to access the member x
26 cout << fmt::format(" this->x = {}\n",
27
28 // explicitly use the dereferenced this pointer and the dot operator
29 // to access the member x
30 cout << fmt::format("(*this).x = {}\n",
31 }
32
33 int main() {
34     const Test testObject{12}; // instantiates the object
35     testObject.print();
36 }
```

< **>**

```

x = 12
this->x = 12
(*this).x = 12

```

Fig. 9.29 Using the `this` pointer to refer to object members.

For illustration purposes, member function `print` (lines 20–31) first displays `x` using the `this` pointer implicitly (line 22)—only the data member’s name is specified. Then `print` uses two different notations to access `x` through the `this` pointer:

- `this->m_x` (line 26) and
- `(*this).m_x` (line 30).

The parentheses around `*this` (line 30) are required because the dot operator (`.`) has higher precedence than the `*` pointer-dereferencing operator. Without the parentheses, the expression `*this.x` would be evaluated as if it were parenthesized as `* (this.x)`. The dot operator cannot be used with a pointer, so this would be a compilation error.

9.19.2 Using the `this` Pointer to Enable Cascaded Function Calls

 **SE** Another use of the `this` pointer is to enable **cascaded member-function calls**—that is, invoking multiple functions sequentially in the same statement, as you’ll see in line 11 of Fig. 9.32. The program of Figs. 9.30–9.32 modifies class `Time`’s `setTime`, `setHour`, `setMinute` and `setSecond` functions such that each returns a reference to the `Time` object on which it’s called. This reference enables cascaded member-function calls. In Fig. 9.31, the last statement in each of these member functions’ bodies returns a reference to `*this` (lines 19, 29, 39 and 49).

[Click here to view code image](#)

```
1 // Fig. 9.30: Time.h
2 // Time class modified to enable cascaded m·
3 #pragma once // prevent multiple inclusions
4 #include <string>
5
6 class Time {
7 public:
8     // default constructor because it can be
9     explicit Time(int hour = 0, int minute =
10
11    // set functions
12    Time& setTime(int hour, int minute, int s);
13    Time& setHour(int hour); // set hour (af
14    Time& setMinute(int minute); // set minu
15    Time& setSecond(int second); // set seco
16
17    int getHour() const; // return hour
18    int getMinute() const; // return minute
19    int getSecond() const; // return second
20    std::string to24HourString() const; // 2
21    std::string to12HourString() const; // 1
22 private:
```

```
23     int m_hour{0}; // 0 - 23 (24-hour clock)
24     int m_minute{0}; // 0 - 59
25     int m_second{0}; // 0 - 59
26 }
```



Fig. 9.30 Time class modified to enable cascaded member-function calls.

[Click here to view code image](#)

```
1 // Fig. 9.31: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept>
4 #include <fmt/format.h> // In C++20, this w
5 #include "Time.h" // Time class definition
6 using namespace std;
7
8 // Time constructor initializes each data m
9 Time::Time(int hour, int minute, int second
10             setHour(hour); // validate and set priva
11             setMinute(minute); // validate and set p
12             setSecond(second); // validate and set p
13 }
14
15 // set new Time value using 24-hour time
16 Time& Time::setTime(int hour, int minute, int
17                      Time time{hour, minute, second}; // crea
18                      *this = time; // if time is valid, assig
19                      return *this; // enables cascading
20 }
21
22 // set hour value
23 Time& Time::setHour(int hour) { // note Tim
24     if (hour < 0 || hour >= 24) {
25         throw invalid_argument{"hour must be
```

```
26     }
27
28     m_hour = hour;
29     return *this; // enables cascading
30 }
31
32 // set minute value
33 Time& Time::setMinute(int m) { // note Time
34     if (m < 0 || m >= 60) {
35         throw invalid_argument{"minute must be between 0 and 59"};
36     }
37
38     m_minute = m;
39     return *this; // enables cascading
40 }
41
42 // set second value
43 Time& Time::setSecond(int s) { // note Time
44     if (s < 0 || s >= 60) {
45         throw invalid_argument{"second must be between 0 and 59"};
46     }
47
48     m_second = s;
49     return *this; // enables cascading
50 }
51
52 // get hour value
53 int Time::getHour() const {return m_hour;}
54
55 // get minute value
56 int Time::getMinute() const {return m_minute;}
57
58 // get second value
59 int Time::getSecond() const {return m_second;}
60
61 // return Time as a string in 24-hour format
62 string Time::to24HourString() const {
```

```
63     return fmt::format("{:02d}:{:02d}:{:02d}\n"
64             getHour(), getMinute(), getSecond()
65     }
66
67 // return Time as string in 12-hour format
68 string Time::to12HourString() const {
69     return fmt::format("{}:{}{:02d} {}"
70         ((getHour() % 12 == 0) ? 12 : getHour(),
71          getMinute(), getSecond(), (getHour() - 12) % 12)
72 }
```



Fig. 9.31 Time class member-function definitions modified to enable cascaded member-function calls.

[Click here to view code image](#)

```
1 // fig09_32.cpp
2 // Cascading member-function calls with the
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will
5 #include "Time.h" // Time class definition
6 using namespace std;
7
8 int main() {
9     Time t{}; // create Time object
10
11     t.setHour(18).setMinute(30).setSecond(22)
12
13     // output time in 24-hour and 12-hour formats
14     cout << fmt::format("24-hour time: {}\n12-hour time: {}"
15                         t.to24HourString(), t.to12HourString())
16
17     // cascaded function calls
18     cout << fmt::format("New 12-hour time: {}"
19                         t.setTime(20, 20, 20).to12HourString())
```

```
20 }
```

```
< ━━━━ >  
24-hour time: 18:30:22  
12-hour time: 6:30:22 PM  
New 12-hour time: 8:20:20 PM
```

Fig. 9.32 Cascading member-function calls with the `this` pointer.

Notice the elegant new implementation of member function `setTime`. Line 17 in Fig. 9.31, creates a local `Time` object called `time` using `setTime`'s arguments. While initializing this object, the constructor call will fail and throw an exception if any argument is out of range. Otherwise, `setTime`'s arguments are all valid, so line 18 assigns the `time` object's members to `*this`—the `Time` object on which `setTime` was called.

In the next program (Fig. 9.32), we create `Time` object `t` (line 9), then use it in cascaded member-function calls (lines 11 and 19).

Why does the technique of returning `*this` as a reference work? The dot operator (`.`) groups left-to-right, so line 11

[Click here to view code image](#)

```
t.setHour(18).setMinute(30).setSecond(22);
```

first evaluates `t.setHour(18)`, which returns a reference to (the updated) object `t` as the value of this function call. The remaining expression is then interpreted as

[Click here to view code image](#)

```
t.setMinute(30).setSecond(22);
```

The `t.setMinute(30)` call executes and returns a reference to the (further updated) object `t`. The remaining expression is interpreted as

```
t.setSecond(22);
```

Line 19 (Fig. 9.32) also uses cascading. Note that we cannot chain another `Time` member-function call after `to12HourString`, because it does not

return a reference to a Time object. However, we could chain a call to a string member function, because `to12HourString` returns a string. Chapter 11 presents several practical examples of using cascaded function calls.

9.20 static Class Members—Classwide Data and Member Functions

There is an important exception to the rule that each object of a class has its own copy of all the class's data members. In certain cases, all objects of a class should share only one copy of a variable. A **static data member** is used for these and other reasons. Such a variable represents "classwide" information—that is, data shared by all objects of the class. You can use static data members to save storage when a single copy of the data for all objects of a class will suffice, such as a constant that can be shared by all objects of the class.

Motivating Classwide Data

Let's further motivate the need for static classwide data with an example. Suppose that we have a video game with Martians and other space creatures. Each Martian tends to be brave and willing to attack other space creatures when the Martian is aware that at least five Martians are present. If fewer than five are present, each Martian becomes cowardly. So each Martian needs to know the `martianCount`. We could endow each object of class `Martian` with `martianCount` as a data member. If we do, every Martian will have its own copy of the data member. Every time we create a new Martian, we'd have to update the data member `martianCount` in all `Martian` objects. Doing this would require every `Martian` object to know about all other `Martian` objects in memory. This wastes space with redundant `martianCount` copies and wastes time in updating the separate copies. Instead, we declare `martianCount` to be static to make it classwide data. Every `Martian` can access `martianCount` as if it were a data member of the `Martian`, but only one copy of the static variable `martianCount` is maintained in the program. This saves space. We have the `Martian` constructor increment static variable `martianCount` and the `Martian` destructor decrement

`martianCount`. Because there's only one copy, we do not have to increment or decrement separate copies of `martianCount` for every `Martian` object.

Scope and Initialization of `static` Data Members

17 A class's `static` data members have class scope. A `static` data member must be initialized exactly once. Fundamental-type `static` data members are initialized by default to 0. A `static const` data member can have an in-class initializer. As of C++17, you also may use in-class initializers for a non-`const` `static` data member by preceding its declaration with the `inline` keyword (as you'll see momentarily in Fig. 9.33). If a `static` data member is an object of a class that provides a default constructor, the `static` data member need not be explicitly initialized because its default constructor will be called.

[Click here to view code image](#)

```
1 // Fig. 9.33: Employee.h
2 // Employee class definition with a static :
3 // track the number of Employee objects in :
4 #pragma once
5 #include <string>
6 #include <string_view>
7
8 class Employee {
9 public:
10    Employee(std::string_view firstName, std
11    ~Employee(); // destructor
12    const std::string& getFirstName() const;
13    const std::string& getLastName() const;
14
15    // static member function
16    static int getCount(); // return # of ob
17 private:
18    std::string m(firstName;
19    std::string m(lastName;
```

```
20
21     // static data
22     inline static int m_count{0}; // number
23 }
```



Fig. 9.33 Employee class definition with a `static` data member to track the number of Employee objects in memory.

Accessing `static` Data Members

A class's `static` members exist even when no objects of that class exist. To access a public `static` class data member or member function, simply prefix the class name and the scope resolution operator (`::`) to the member name. For example, if our `martianCount` variable is `public`, it can be accessed with `Martian::martianCount`, even when there are no `Martian` objects. (Of course, using `public` data is discouraged.)

 **SE** A class's `private` (and `protected`; [Chapter 10](#)) `static` members are normally accessed through the class's `public` member functions or `friends`. To access a `private` `static` or `protected` `static` data member when no objects of the class exist, provide a `public` **static member function** and call the function by prefixing its name with the class name and scope resolution operator. A `static` member function is a service of the class as a whole, not of a specific object of the class.

Demonstrating `static` Data Members

This example demonstrates a `private` `inline` `static` data member called `m_count` (Fig. 9.33, line 22), which is initialized to 0, and a `public` `static` member function called `getCount` (Fig. 9.33, line 16). A `static` data member also can be initialized at file scope in the class's implementation file. For instance, we could have placed the following statement in `Employee.cpp` (Fig. 9.34) after the `Employee.h` header is included:

```
int Employee::count{0};
```

[Click here to view code image](#)

```
1 // Fig. 9.34: Employee.cpp
2 // Employee class member-function definitions
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be part of the standard library
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // define static member function that returns the count of Employee objects
9 // instantiated (declared)
10 int Employee::getCount() {return m_count;}
11
12 // constructor initializes non-static data members
13 // increments static data member count
14 Employee::Employee(string_view firstName, string_view lastName)
15     : m(firstName), m(lastName), m_count(0) {
16     ++m_count; // increment static count of Employee objects
17     cout << fmt::format("Employee constructor called for {} {}, count = {}",
18                         m.firstName, m.lastName);
19 }
20
21 // destructor decrements the count
22 Employee::~Employee() {
23     cout << fmt::format("~Employee() called for {} {}, count = {}",
24                         m.firstName, m.lastName);
25     --m_count; // decrement static count of Employee objects
26 }
27
28 // return first name of employee
29 const string& Employee::getFirstName() const {
30
31 // return last name of employee
32 const string& Employee::getLastName() const {
```



Fig. 9.34 Employee class member-function definitions.

In Fig. 9.34, line 10 defines static member function getCount. Note that line 10 does not include the `static` keyword, which cannot be applied to a member definition that appears outside the class definition. In this program, data member `m_count` maintains a count of the number of `Employee` objects in memory at a given time. When `Employee` objects exist, member `m_count` can be referenced through any member function of an `Employee` object. In Fig. 9.34, `m_count` is referenced by both line 16 in the constructor and line 25 in the destructor.

Figure 9.35 uses static member function `getCount` to determine the number of `Employee` objects in memory at various points in the program. The program calls `Employee::getCount()`:

- before any `Employee` objects have been created (line 12),
- after two `Employee` objects have been created (line 23) and
- after those `Employee` objects have been destroyed (line 33).

[Click here to view code image](#)

```
1 // fig09_35.cpp
2 // static data member tracking the number o
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this w
5 #include "Employee.h" // Employee class def
6 using namespace std;
7
8 int main() {
9     // no objects exist; use class name and .
10    // operator to access static member func
11    cout << fmt::format("Initial employee co
12                Employee::getCount()); // use
13
14    // the following scope creates and destro
15    // Employee objects before main terminat
16    {
17        const Employee e1{"Susan", "Baker"};
18        const Employee e2{"Robert", "Jones"};
```

```

19
20      // two objects exist; call static mem
21      // using the class name and the scope
22      cout << fmt::format("Employee count a
23                      Employee::getCount());
24
25      cout << fmt::format("Employee 1: {} {
26                      e1.getFirstName(), e1.getL
27                      e2.getFirstName(), e2.getL
28      }
29
30      // no objects exist, so call static memb
31      // using the class name and the scope re
32      cout << fmt::format("Employee count afte
33                      Employee::getCount());
34 }
```

< >

```

Initial employee count: 0
Employee constructor called for Susan Baker
Employee constructor called for Robert Jones
Employee count after creating objects: 2

Employee 1: Susan Baker
Employee 2: Robert Jones
~Employee() called for Robert Jones
~Employee() called for Susan Baker
Employee count after objects are deleted: 0
```

Fig. 9.35 static data member tracking the number of objects of a class.

Lines 16–28 in main define a nested scope. Recall that local variables exist until the scope in which they’re defined terminates. In this example, we create two Employee objects in the nested scope (lines 17–18). As each constructor executes, it increments class Employee’s static data member

count. These Employee objects are destroyed when the program reaches line 28. At that point, each object's destructor executes and decrements class Employee's static data member count.

static Member Function Notes

 **SE** A member function should be declared static if it does not access the class's non-static data members or non-static member functions. A static member function does not have a this pointer, because static data members and static member functions exist independently of any objects of a class. The this pointer must refer to a specific object, but a static member function can be called when there are no objects of its class in memory. So, using the this pointer in a static member function is a compilation error.

 **ERR**  **ERR** A static member function may not be declared const. The const qualifier indicates that a function cannot modify the contents of the object on which it operates, but static member functions exist and operate independently of any objects of the class. So, declaring a static member function const is a compilation error.

20 9.21 Aggregates in C++20

According to Section 9.4.1 of the C++ standard document

<http://wg21.link/n4861>

an **aggregate type** is a built-in array, an array object or an object of a class that:

- does not have user-declared constructors,
- does not have private or protected (Chapter 10) non-static data members,
- does not have virtual functions (Chapter 10) and
- does not have virtual (Chapter 19), private (Chapter 10) or protected (Chapter 10) base classes.

20 The requirement for no user-declared constructors was a C++20 change to the definition of aggregates. It prevents a case in which initializing an aggregate object could circumvent calling a user-declared constructor.²⁸

28. “Prohibit aggregates with user-declared constructors.” Accessed July 12, 2020. <http://wg21.link/p1008r1>.

You can define an aggregate using a class in which all the data is declared public. However, a **struct** is a class that contains only public members by default. The following struct defines an aggregate type named Record containing four public data members:

[Click here to view code image](#)

```
struct Record {  
    int account;  
    string first;  
    string last;  
    double balance;  
};
```

 **CG** The C++ Core Guidelines recommend using class rather than struct if any data member or member function needs to be non-public.²⁹

29. C++ Core Guidelines. Accessed July 12, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-class>.

9.21.1 Initializing an Aggregate

You can initialize an object of aggregate type Record as follows:

[Click here to view code image](#)

```
Record record{100, "Brian", "Blue", 123.45};
```

11 In C++11, you could not use a list initializer for an aggregate-type object if any of the type’s non-static data-member declarations contained in-class initializers. For example, the initialization above would have generated a compilation error if the aggregate type Record were defined with a default value for balance, as in:

[Click here to view code image](#)

```
struct Record {  
    int account;  
    string first;  
    string last;  
    double balance{0.0};  
};
```

14 C++14 removed this restriction. Also, if you initialize an aggregate-type object with fewer initializers than there are data members in the object, as in

[Click here to view code image](#)

```
Record record{0, "Brian", "Blue"};
```

the remaining data members are initialized as follows:

- Data members with in-class initializers use those values—in the preceding case, `record`'s `balance` is set to 0.0.
- Data members without in-class initializers are initialized with empty braces ({}). Empty-brace initialization sets fundamental-type variables to 0, sets bools to `false` and **value initializes** objects—that is, they're zero initialized, then the default constructor is called for each object.

20 9.21.2 C++20: Designated Initializers

As of C++20, aggregates now support **designated initializers** in which you can specify which data members to initialize by name. Using the preceding `struct Record` definition, we could initialize a `Record` object as follows:

[Click here to view code image](#)

```
Record record{.first{"Sue"}, .last{"Green"}};
```

explicitly initializing only a subset of the data members. Each explicitly named data member is preceded by a dot (.), and the identifiers that you specify must be listed in the same order as they're declared in the aggregate type. The preceding statement initializes the data members `first` and `last` to "Sue" and "Green", respectively. The remaining data members get

their default initializer values:

- account is set to 0 and
- balance is set to its default value in the type definition—in this case, 0.0.

Other Benefits of Designated Initializers³⁰

³⁰. “Designated Initialization.” Accessed July 12, 2020. <http://wg21.link/p0329r0>.

 SE Adding new data members to an aggregate type will not break existing statements that use designated initializers. Any new data members that are not explicitly initialized simply receive their default initialization. Designated initializers also improve compatibility with the C programming language, which has had this feature since C99.

9.22 Objects Natural Case Study: Serialization with JSON

More and more computing today is done “in the cloud”—that is, distributed across the Internet. Many applications you use daily communicate over the Internet with **cloud-based services** that use massive clusters of computing resources (computers, processors, memory, disk drives, databases, etc.).

A service that provides access to itself over the Internet is known as a **web service**. Applications typically communicate with web services by sending and receiving JSON objects. **JSON (JavaScript Object Notation)** is a text-based, human-and-computer-readable, data-interchange format that represents objects as collections of name–value pairs. JSON has become the preferred data format for transmitting objects across platforms.

JSON Data Format

Each JSON object contains a comma-separated list of property names and values in curly braces. For example, the following name–value pairs might represent a client record:

[Click here to view code image](#)

```
{"account": 100, "name": "Jones", "balance": 24.9}
```



JSON also supports arrays as comma-separated values in square brackets. For example, the following represents a JSON array of numbers:

```
[100, 200, 300]
```

Values in JSON objects and arrays can be:

- strings in double-quotes (like "Jones"),
- numbers (like 100 or 24.98),
- JSON Boolean values (represented as true or false),
- null (to represent no value),
- arrays of any valid JSON value, and
- other JSON objects.

JSON arrays may contain elements of the same or different types.

Serialization

Converting an object into another format for storage or transmission over the Internet is known as **serialization**. Similarly, reconstructing an object from serialized data is known as **deserialization**. JSON is just one of several serialization formats. Other common formats include binary data and XML (eXtensible Markup Language).

SEC Serialization Security

Some programming languages have their own serialization mechanisms that use a language-native format. Deserializing objects using these native serialization formats is a source of various security issues. According to the Open Web Application Security Project (OWASP), these native mechanisms “can be repurposed for malicious effect when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks.”³¹ OWASP also indicates that you can significantly reduce attack risk by avoiding language-native serialization formats in favor of “pure data” formats like JSON or XML.

³¹. “Deserialization Cheat Sheet.” OWASP Cheat Sheet Series. Accessed July 18, 2020. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Che

cereal Header-Only Serialization Library

The **cereal header-only library**³² serializes objects to and deserializes objects from JSON (which we'll demonstrate), XML or binary formats. The library supports fundamental types and can handle most standard library types if you include each type's appropriate cereal header. As you'll see in the next section, cereal also supports custom types. The cereal documentation is available at:

32. Copyright (c) 2017, Grant, W. Shane and Voorhies, Randolph. cereal—A C++11 library for serialization. URL: <http://uscilab.github.io/cereal/>. All rights reserved.

<https://uscilab.github.io/cereal/index.html>

We've included the cereal library for your convenience in the libraries folder with the book's examples. You must point your IDE or compiler at the library's include folder, as you've done in several earlier Objects Natural case studies.

9.22.1 Serializing a `vector` of Objects containing public Data

Let's begin by serializing objects containing only public data. In Fig. 9.36, we:

- create a vector of Record objects and display its contents,
- use cereal to serialize it to a text file, then
- deserialize the file's contents into a vector of Record objects and display the serialized Records.

[Click here to view code image](#)

```
1 // fig09_36.cpp
2 // Serializing and deserializing objects with cereal
3 #include <iostream>
4 #include <fstream>
5 #include <vector>
6 #include <fmt/format.h> // In C++20, this will be <format>
7 #include <cereal/archives/json.hpp>
```

```
8 #include <cereal/types/vector.hpp>
9
10 using namespace std;
```



Fig. 9.36 Serializing and deserializing objects with the cereal library.

To perform JSON serialization, include the cereal header json.hpp (line 7). To serialize a std::vector, include the cereal header vector.hpp (line 8).

Aggregate Type Record

Record is an aggregate type defined as a struct. Recall that an aggregate's data must be public, which is the default for a struct definition:

[Click here to view code image](#)

```
11
12 struct Record {
13     int account{};
14     string first{};
15     string last{};
16     double balance{};
17 };
18
```

Fig. 9.37

Function `serialize` for Record Objects

The cereal library allows you to designate how to perform serialization several ways. If the types you wish to serialize have all public data, you can simply define a function template `serialize` (lines 21–27) that receives an `Archive` as its first parameter and an object of your type as the second.³³ This function is called both for serializing and deserializing the Record objects. Using a function template enables you to choose among

serialization and deserialization using JSON, XML or binary formats by passing an object of the appropriate cereal archive type. The library provides archive implementations for each case.

33. Function `serialize` also may be defined as a member function template with only an Archive parameter. For details, see https://uscilab.github.io/cereal/serialization_functions.html.

[Click here to view code image](#)

```
19 // function template serialize is responsible
20 // deserializing Record objects to/from the sp
21 template <typename Archive>
22 void serialize(Archive& archive, Record& record)
23     archive(cereal::make_nvp("account", record.
24         cereal::make_nvp("first", record.first),
25         cereal::make_nvp("last", record.last),
26         cereal::make_nvp("balance", record.balan
27     )
28 }
```

Each cereal archive type has an overloaded parentheses operator that enables you to use parameter archive objects as function names, as shown with parameter `archive` in lines 23–26. Depending on whether you’re serializing or deserializing a Record, this function will either:

- output the contents of the Record to a specified stream or
- input previously serialized data from a specified stream and create a Record object.

Each call to `cereal::make_nvp` (that is, “make name–value pair”), like line 23

[Click here to view code image](#)

```
cereal::make_nvp("account", record.account)
```

is primarily for the serialization step. It makes a name–value pair with the name in the first argument (in this case, "account") and the value in the second argument (in this case, the `int` value `record.account`). Naming

the values is not required but makes the JSON output more readable as you'll soon see. Otherwise, cereal uses names like `value0`, `value1`, etc.

Function `displayRecords`

We provide function `displayRecords` to show you the contents of our Record objects before serialization and after deserialization. The function simply displays the contents of each Record in the vector it receives as an argument:

[Click here to view code image](#)

```
29 // display record at command line
30 void displayRecords(const vector<Record>& records) {
31     for (const auto& r : records) {
32         cout << fmt::format("{} {} {} {:.2f}\n",
33                             r.account, r.first, r.last, r.value);
34     }
35 }
36
```



Creating Record Objects to Serialize

Lines 38–41 in `main` create a vector and initialize it with two Records (lines 39 and 40). The compiler determines the vector's Record element type from the initializers. Line 44 outputs the vector's contents to confirm that the two Records were initialized properly:

[Click here to view code image](#)

```
37 int main() {
38     vector<Record> records{
39         Record{100, "Brian", "Blue", 123.45},
40         Record{200, "Sue", "Green", 987.65}
41     };
42
43     cout << "Records to serialize:\n";
44     displayRecords(records);
```

```
Records to serialize:  
100 Brian Blue 123.45  
200 Sue Green 987.65
```

Serializing Record Objects with cereal::JSONOutputArchive

A `cereal::JSONOutputArchive` serializes data in JSON format to a specified stream, such as the standard output stream or a stream representing a file. Line 47 attempts to open the file `records.json` for writing. If successful, line 48 creates a `cereal::JSONOutputArchive` object named `archive` and initializes it with the output `ofstream` object so `archive` can write the JSON data into a file. Line 49 uses the `archive` object to output a name–value pair with the name "records" and the vector of `Records` as its value. Part of serializing the vector is serializing each of its elements. So line 49 also results in one call to `serialize` (lines 21–27) for each `Record` object in the vector.

[Click here to view code image](#)

```
46     // serialize vector of Records to JSON and  
47     if (ofstream output{"records.json"}) {  
48         cereal::JSONOutputArchive archive{output};  
49         archive(cereal::make_nvp("records", reco  
50     }  
51
```



Contents of records.json

After line 49 executes, the file `records.json` contains the following JSON data:

```

{
    "records": [
        {
            "account": 100,
            "first": "Brian",
            "last": "Blue",
            "balance": 123.45
        },
        {
            "account": 200,
            "first": "Sue",
            "last": "Green",
            "balance": 987.65
        }
    ]
}

```

The outer braces represent the entire JSON document. The darker box highlights the document's one name–value pair named "records", which has as its value a JSON array containing the two JSON objects in the lighter boxes. The JSON array represents the vector records that we serialized in line 49. Each of the JSON objects in the array contains one Record's four name–value pairs that were serialized by the serialize function.

Deserializing Record Objects with cereal::JSONInputArchive

Next, let's deserialize the data and use it to fill a separate vector of Record objects. For cereal to recreate objects in memory, it must have access to each type's default constructor. It will use that to create an object, then directly access that object's data members to place the data into the object. Like classes, the compiler provides a public default constructor for structs if you do not define a custom constructor.

[Click here to view code image](#)

```

52     // deserialize JSON from text file into vec
53     if (ifstream input{"records.json"}) {
54         cereal::JSONInputArchive archive{input};
55         vector<Record> deserializedRecords{ };
56         archive(deserializedRecords); // deseria

```

```
57         cout << "\nDeserialized records:\n";
58         displayRecords(deserializedRecords);
59     }
60 }
```

```
< ━━━━━━ >
Deserialized records:
100 Brian Blue 123.45
200 Sue Green 987.65
```

A `cereal::JSONInputArchive` deserializes data in JSON format from a specified stream, such as the standard input stream or a stream representing a file. Line 53 attempts to open the file `records.json` for reading. If successful, line 54 creates the object archive of type `cereal::JSONInputArchive` and initializes it with the input `ifstream` object so archive can read JSON data from the `records.json` file. Line 55 creates an empty vector of `Records` into which we'll read the JSON data. Line 56 uses the archive object to deserialize the file's data into the `deserializedRecords` object. Part of deserializing the vector is deserializing its elements. This again results in calls to `serialize` (lines 21–27), but because `archive` is a `cereal::JSONInputArchive`, each call to `serialize` reads one `Record`'s JSON data, creates a `Record` object then inserts the data into it.

9.22.2 Serializing a `vector` of Objects containing private Data

It is also possible to serialize objects containing private data. To do so, you must declare the `serialize` function as a friend of the class, so it can access the class's private data. To demonstrate serializing private data, we created a copy of Fig. 9.36 and replaced the aggregate `Record` definition with the class `Record` definition (lines 14–37) in Fig. 9.38.

[Click here to view code image](#)

```
1 // fig09_37.cpp
```

```
2 // Serializing and deserializing objects
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <string_view>
7 #include <vector>
8 #include <fmt/format.h> // In C++20, this will be replaced by std::format
9 #include <cereal/archives/json.hpp>
10 #include <cereal/types/vector.hpp>
11
12 using namespace std;
13
14 class Record {
15     // declare serialize as a friend for direct access
16     template<typename Archive>
17     friend void serialize(Archive& archive, Record& record);
18
19 public:
20     // constructor
21     explicit Record(int account = 0, string_view first = "", string_view last = "", double balance = 0.0)
22         : m_account{account}, m_first{first}, m_last{last}, m_balance{balance} {}
23
24     // get member functions
25     int getAccount() const {return m_account;}
26     const string& getFirst() const {return m_first;}
27     const string& getLast() const {return m_last;}
28     double getBalance() const {return m_balance;}
29
30     // set member functions
31     void setAccount(int account) {m_account = account;}
32     void setFirst(string first) {m_first = first;}
33     void setLast(string last) {m_last = last;}
34     void setBalance(double balance) {m_balance = balance;}
35
36     // output operator
37     friend std::ostream& operator<< (std::ostream& os, Record& record) {
38         os << "Account: " << record.getAccount() << ", First: " << record.getFirst() << ", Last: " << record.getLast() << ", Balance: " << record.getBalance();
39         return os;
40     }
41
42     // input operator
43     friend std::istream& operator>> (std::istream& is, Record& record) {
44         is >> record.m_account >> record.m_first >> record.m_last >> record.m_balance;
45         return is;
46     }
47
48     // equality operator
49     friend bool operator== (const Record& record1, const Record& record2) {
50         return record1.getAccount() == record2.getAccount() &&
51                record1.getFirst() == record2.getFirst() &&
52                record1.getLast() == record2.getLast() &&
53                record1.getBalance() == record2.getBalance();
54     }
55
56     // inequality operator
57     friend bool operator!= (const Record& record1, const Record& record2) {
58         return !operator==(record1, record2);
59     }
60
61     // less than operator
62     friend bool operator< (const Record& record1, const Record& record2) {
63         if (record1.getAccount() != record2.getAccount())
64             return record1.getAccount() < record2.getAccount();
65         if (record1.getFirst() != record2.getFirst())
66             return record1.getFirst() < record2.getFirst();
67         if (record1.getLast() != record2.getLast())
68             return record1.getLast() < record2.getLast();
69         if (record1.getBalance() != record2.getBalance())
70             return record1.getBalance() < record2.getBalance();
71         return false;
72     }
73
74     // greater than operator
75     friend bool operator> (const Record& record1, const Record& record2) {
76         if (record1.getAccount() != record2.getAccount())
77             return record1.getAccount() > record2.getAccount();
78         if (record1.getFirst() != record2.getFirst())
79             return record1.getFirst() > record2.getFirst();
80         if (record1.getLast() != record2.getLast())
81             return record1.getLast() > record2.getLast();
82         if (record1.getBalance() != record2.getBalance())
83             return record1.getBalance() > record2.getBalance();
84         return false;
85     }
86
87     // less than or equal to operator
88     friend bool operator<= (const Record& record1, const Record& record2) {
89         if (record1.getAccount() != record2.getAccount())
90             return record1.getAccount() < record2.getAccount();
91         if (record1.getFirst() != record2.getFirst())
92             return record1.getFirst() < record2.getFirst();
93         if (record1.getLast() != record2.getLast())
94             return record1.getLast() < record2.getLast();
95         if (record1.getBalance() != record2.getBalance())
96             return record1.getBalance() < record2.getBalance();
97         return true;
98     }
99
100    // greater than or equal to operator
101    friend bool operator>= (const Record& record1, const Record& record2) {
102        if (record1.getAccount() != record2.getAccount())
103            return record1.getAccount() > record2.getAccount();
104        if (record1.getFirst() != record2.getFirst())
105            return record1.getFirst() > record2.getFirst();
106        if (record1.getLast() != record2.getLast())
107            return record1.getLast() > record2.getLast();
108        if (record1.getBalance() != record2.getBalance())
109            return record1.getBalance() > record2.getBalance();
110        return true;
111    }
112}
```

```
39 // function template serialize is responsib
40 // deserializing Record objects to/from the
41 template <typename Archive>
42 void serialize(Archive& archive, Record& re
    archive(cereal::make_nvp("account", record.
43     cereal::make_nvp("first", record.m_fi
44     cereal::make_nvp("last", record.m_las
45     cereal::make_nvp("balance", record.m_
46 }
47
48
49 // display record at command line
50 void displayRecords(const vector<Record>& r
51     for (auto& r : records) {
52         cout << fmt::format("{} {} {} {:.2f}\n",
53                             r.getFirst(), r.getLast(),
54
55 }
56
57 int main() {
58     vector<Record> records{
59         Record{100, "Brian", "Blue", 123.45},
60         Record{200, "Sue", "Green", 987.65}
61     };
62
63     cout << "Records to serialize:\n";
64     displayRecords(records);
65
66     // serialize vector of Records to JSON a
67     if (ofstream output{"records2.json"}) {
68         cereal::JSONOutputArchive archive{out
69         archive(cereal::make_nvp("records", r
70     }
71
72     // deserialize JSON from text file into
73     if (ifstream input{"records2.json"}) {
74         cereal::JSONInputArchive archive{inpu
75         vector<Record> serializedRecords{;
```

```
76     archive(deserializedRecords); // dese
77     cout << "\nDeserialized records:\n";
78     displayRecords(deserializedRecords);
79 }
80 }
```

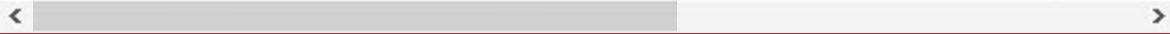


Fig. 9.38 Serializing and deserializing objects containing private data.

This Record class provides a constructor (21–24), *get* member functions (lines 27–30) and private data (lines 33–36). There are two key items to note about this class:

- Lines 16–17 declare the function template `serialize` as a friend of this class. This enables `serialize` to access directly the data members `account`, `first`, `last` and `balance`.
- The constructor's parameters all have default arguments, which allows `cereal` to use this as the default constructor when deserializing `Record` objects.

The `serialize` function (lines 41–46) now accesses class `Record`'s private data members, and the `displayRecords` function (lines 50–55) now uses each `Record`'s *get* functions to access the data to display. The main function is identical to section [Section 9.22.1](#) and produces the same results, so we do not show the output here.

9.23 Wrap-Up

In this chapter, you created your own classes, created objects of those classes and called member functions of those objects to perform useful actions. You declared data members of a class to maintain data for each object of the class, and you defined member functions to operate on that data. You also learned how to use a class's constructor to specify the initial values for an object's data members.

We used a `Time` class case study to introduce various additional features. We showed how to engineer a class to separate its interface from its implementation. You used the arrow operator to access an object's members

via a pointer to an object. You saw that member functions have class scope—the member function’s name is known only to the class’s other member functions unless referred to by a client of the class via an object name, a reference to an object of the class, a pointer to an object of the class or the scope resolution operator. We also discussed access functions (commonly used to retrieve the values of data members or to test whether a condition is *true* or *false*), and utility functions (`private` member functions that support the operation of the class’s `public` member functions).

You saw that a constructor can specify default arguments that enable it to be called multiple ways. You also saw that any constructor that can be called with no arguments is a default constructor and that there can be at most one default constructor per class. We demonstrated how to share code among constructors with delegating constructors. We discussed destructors for performing termination housekeeping on an object before that object is destroyed, and demonstrated the order in which an object’s constructors and destructors are called.

We showed the problems that can occur when a member function returns a reference or a pointer to a `private` data member, which breaks the class’s encapsulation. We also showed that objects of the same type can be assigned to one another using the default assignment operator.

You learned how to specify `const` objects and `const` member functions to prevent modifications to objects, thus enforcing the principle of least privilege. You also learned that, through composition, a class can have objects of other classes as members. We demonstrated how to declare and use `friend` functions.

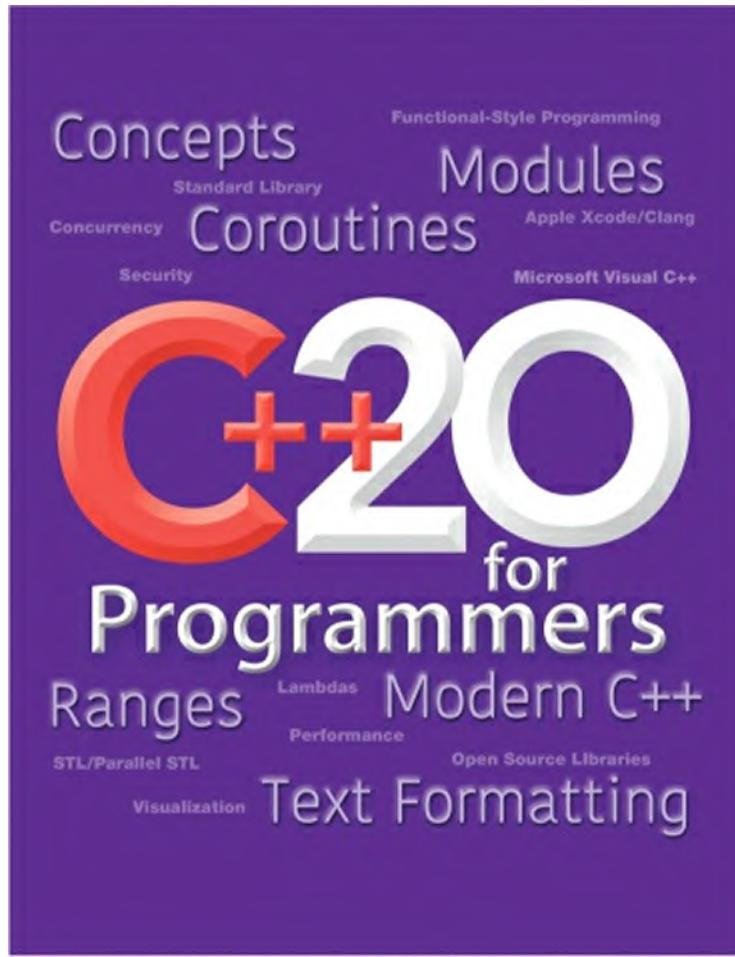
You saw that the `this` pointer is passed as an implicit argument to each of a class’s non-`static` member functions, allowing them to access the correct object’s data members and other non-`static` member functions. We used the `this` pointer explicitly to access the class’s members and to enable cascaded member-function calls. We motivated the notion of `static` data members and member functions and demonstrated how to declare and use them.

We introduced aggregate types and C++20’s designated initializers for aggregates. Finally, we presented our next “objects natural” case study on serializing objects with JSON (JavaScript Object Notation) using the

`cereal` library.

In the next chapter, we continue our discussion of classes by introducing inheritance. We'll see classes that share common attributes and behavior can inherit them from a common "base" class. Then, we build on our discussion of inheritance by introducing polymorphism. This object-oriented concept enables us to write programs that handle, in a more general manner, objects of classes related by inheritance.

Chapter 10. OOP: Inheritance and Runtime Polymorphism



Objectives

In this chapter, you'll:

- Use traditional and modern inheritance idioms, and understand base classes and derived classes.
- Understand the order in which C++ calls constructors and destructors in inheritance hierarchies.
- See how runtime polymorphism can make programming more convenient and systems more easily extensible.

- Use `override` to tell the compiler that a derived-class function overrides a base-class `virtual` function.
 - Use `final` at the end of a function's prototype to indicate that function may not be overridden.
 - Use `final` after a class's name in its definition to indicate that a class cannot be a base class.
 - Perform inheritance with abstract and concrete classes.
 - See how C++ can implement `virtual` functions and dynamic binding—and you'll estimate `virtual` function overhead.
 - Use the non-`virtual` interface idiom (NVI) with `public non-virtual` and `private/protected virtual` functions.
 - Use interfaces to create more flexible runtime-polymorphism systems.
 - Implement runtime polymorphism without class hierarchies via `std::variant` and `std::visit`.
-

Outline

10.1 Introduction

10.2 Base Classes and Derived Classes

10.2.1 CommunityMember Class Hierarchy

10.2.2 Shape Class Hierarchy and `public` Inheritance

10.3 Relationship between Base and Derived Classes

10.3.1 Creating and Using a SalariedEmployee Class

10.3.2 Creating a SalariedEmployee–SalariedCommissionEmployee Inheritance Hierarchy

10.4 Constructors and Destructors in Derived Classes

10.5 Intro to Runtime Polymorphism: Polymorphic Video Game

10.6 Relationships Among Objects in an Inheritance Hierarchy

10.6.1 Invoking Base-Class Functions from Derived-Class Objects

10.6.2 Aiming Derived-Class Pointers at Base-Class Objects
10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers

10.7 Virtual Functions and Virtual Destructors

10.7.1 Why virtual Functions Are Useful
10.7.2 Declaring virtual Functions
10.7.3 Invoking a virtual Function
10.7.4 virtual Functions in the SalariedEmployee Hierarchy
10.7.5 virtual Destructors
10.7.6 final Member Functions and Classes

10.8 Abstract Classes and Pure virtual Functions

10.8.1 Pure virtual Functions
10.8.2 Device Drivers: Polymorphism in Operating Systems

10.9 Case Study: Payroll System Using Runtime Polymorphism

10.9.1 Creating Abstract Base-Class Employee
10.9.2 Creating Concrete Derived-Class SalariedEmployee
10.9.3 Creating Concrete Derived-Class CommissionEmployee
10.9.4 Demonstrating Runtime Polymorphic Processing

10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

10.11 Non-Virtual Interface (NVI) Idiom

10.12 Program to an Interface, Not an Implementation

10.12.1 Rethinking the Employee Hierarchy—CompensationModel Interface
10.12.2 Class Employee
10.12.3 CompensationModel Implementations
10.12.4 Testing the New Hierarchy

10.13 Runtime Polymorphism with std::variant and std::visit

10.14 Multiple Inheritance

10.14.1 Diamond Inheritance
10.14.2 Eliminating Duplicate Subobjects with virtual Base-

Class Inheritance

[10.15 protected Class Members](#)

[10.16 public, protected and private Inheritance](#)

[10.17 Wrap-Up](#)

10.1 Introduction

This chapter continues our object-oriented programming (OOP) discussion by introducing inheritance and runtime polymorphism. With **inheritance**, you'll create classes that absorb existing classes' capabilities, then customize or enhance them.

When creating a class, you can specify that the new class should **inherit** an existing class's members. This existing class is called the **base class**, and the new class is called the **derived class**. Some programming languages, such as Java and C#, use the terms **superclass** and **subclass** for base class and derived class.

Has-a vs. Is-a Relationships

We distinguish between the *has-a* relationship and the *is-a* relationship:

- The *has-a* relationship represents composition ([Section 9.17](#)) in which an object *contains* as members one or more objects of other classes. For example, a *Car* *has a* steering wheel, *has a* brake pedal, *has an* engine, *has a* transmission, etc.
 - The **is-a relationship** represents inheritance. In an *is-a* relationship, a derived-class object also can be treated as an object of its base-class type. For example, a *Car* *is a* *Vehicle*, so a *Car* also exhibits a *Vehicle*'s behaviors and attributes.¹
1. We'll see that a base-class object cannot implicitly be treated as an object of any of its derived classes.

Runtime Polymorphism

We'll explain and demonstrate runtime polymorphism with inheritance hierarchies.² **Runtime polymorphism enables you to conveniently "program in the general" rather than "program in the specific."** Programs can process objects of classes related by inheritance as if they're all

objects of the base-class type. You'll see that runtime polymorphic code—as we'll initially implement it with inheritance and `virtual` functions—refers to objects via base-class pointers or base-class references.

2. We'll see later that runtime polymorphism does not have to be done with inheritance hierarchies, and we'll also discuss compile-time polymorphism.

Implementing for Extensibility

With runtime polymorphism, you can design and implement systems that are more easily **extensible**—new classes can be added with little or no modification to the program's general portions, as long as the new classes are part of the program's inheritance hierarchy. You modify only code that requires direct knowledge of the new classes. Suppose a new class `Car` inherits from class `Vehicle`. We need to write only the new class and code that creates `Car` objects and adds them to the system—the new class simply “plugs right in.” The general code that processes `Vehicles` can remain the same.

Discussion of Runtime Polymorphism with Virtual Functions “Under the Hood”

A key feature of this chapter is its discussion of runtime polymorphism, `virtual` functions and dynamic binding “under the hood.” The C++ standard document does not specify how language features should be implemented. We use a detailed illustration to explain how runtime polymorphism with `virtual` functions *can* be implemented in C++.

This chapter presents a traditional introduction to inheritance and runtime polymorphism to acquaint you with basic-through-intermediate-level thinking and ensure that you understand the mechanics of how these technologies work. Later in this chapter and in the remainder of the book, we'll address current thinking and programming idioms.

Non-Virtual Interface Idiom

Next, we consider another runtime polymorphism approach—the non-virtual interface idiom (NVI), in which each base-class function serves only one purpose:

- as a function that client code can call to perform a task, or
- as a function that derived classes can customize.

The customizable functions are internal implementation details of the classes, making systems easier to maintain and evolve without requiring code changes in client applications.

Interfaces and Dependency Injection

To explain the mechanics of inheritance, our first few examples focus on **implementation inheritance**, which is primarily used to define closely related classes with many of the same data members and member-function implementations. For decades, this style of inheritance was widely practiced in the object-oriented programming community. Over time, though, experience revealed its weaknesses. C++ is used to build real-world, business-critical and mission-critical systems—often at a grand scale. The empirical results from years of building such implementation-inheritance-based systems show that they can be challenging to modify.

So, we'll refactor one of our examples to use **interface inheritance**. The base class will not provide any implementation details. Instead, it will contain “placeholders” that tell derived classes the functions they're required to implement. The derived classes will provide the implementation details—that is, the data members and member-function implementations. As part of this example, we'll introduce **dependency injection** in which a class contains a pointer to an object that provides a behavior required by objects of the class—in our example, calculating an employee's earnings. This pointer can be reaimed—for example, if an employee gets promoted, we can change the earnings calculation by aiming the pointer at an appropriate object. Then, we'll discuss how this refactored example is easier to evolve.

Runtime Polymorphism Via `std::variant` and `std::visit`

17 Next, we'll implement runtime polymorphism for objects of classes that are *not* related by inheritance and do *not* have virtual functions. To accomplish this, we'll use C++17's class template `std::variant` and the standard-library function `std::visit`. **Invoking common functionality on objects of unrelated types is called duck typing.**

Multiple Inheritance

We'll demonstrate **multiple inheritance**, which enables a derived class to inherit the members of several base classes. We discuss potential problems

with multiple inheritance and how virtual inheritance can be used to solve them.

Other Ways to Implement Polymorphism

20 Finally, we'll overview other runtime-polymorphism techniques and several template-based **compile-time polymorphism** approaches. We'll implement some of these in [Chapter 15](#), Templates, C++20 Concepts and Metaprogramming. There you'll see that C++20's new **concepts** feature obviates some older techniques and expands your "toolkit" for creating compile-time, template-based solutions. We'll also provide links to advanced resources for developers who wish to dig deeper.

Chapter Goal

A goal of this chapter is to familiarize you with inheritance and runtime polymorphism mechanics, so you'll be able to better appreciate modern polymorphism idioms and techniques that can promote ease of modifiability and better performance.

10.2 Base Classes and Derived Classes

The following table lists several simple base-class and derived-class examples. Base classes tend to be *more general* and derived classes *more specific*:

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Vehicle	Car, Motorcycle, Boat
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Every derived-class object *is an* object of its base-class type, and one base class can have many derived classes. So, the set of objects a base class represents typically is larger than the set of objects a derived class represents. For example, the base class `Vehicle` represents all vehicles, including cars,

trucks, boats, airplanes, bicycles, etc. By contrast, the derived-class `Car` represents a smaller, more specific subset of all `Vehicles`.

10.2.1 CommunityMember Class Hierarchy

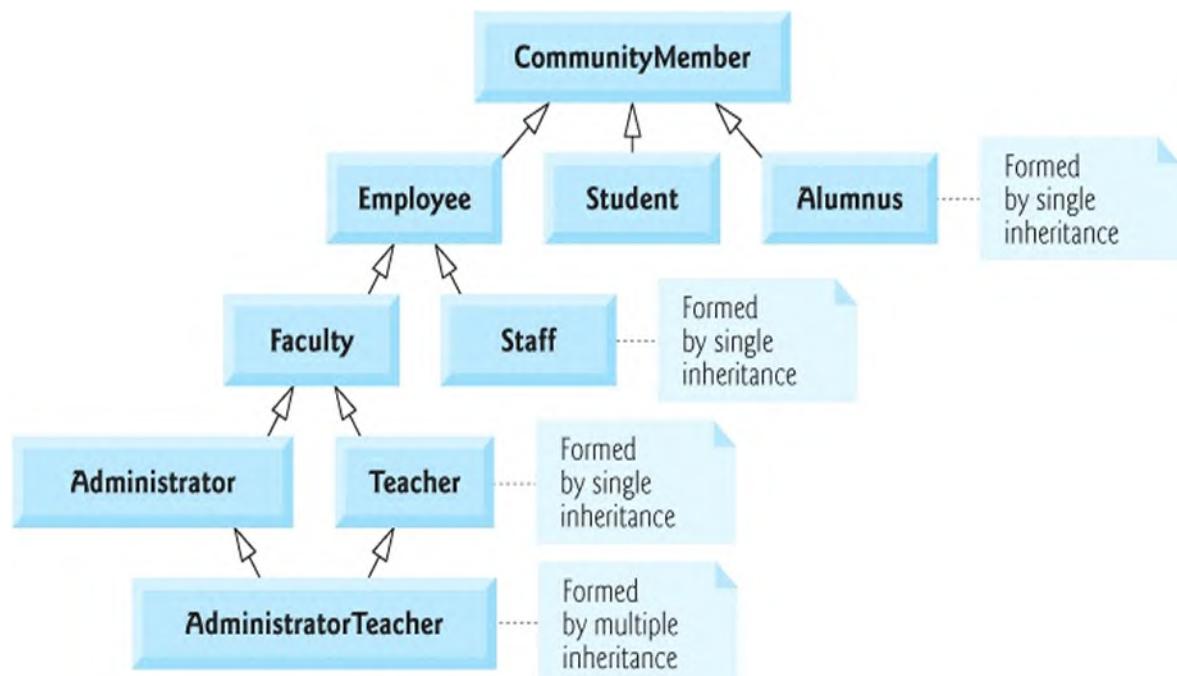
Inheritance relationships naturally form **class hierarchies**. A base class exists in a hierarchical relationship with its derived classes. Once classes are employed in inheritance hierarchies, they become coupled with other classes.³ A class can be

3. We'll see that tightly coupled classes can make systems difficult to modify. We'll show alternatives to avoid tight coupling.

1. a base class that supplies members to other classes,
2. a derived class that inherits members from other classes, or
3. both.⁴

4. Some leaders in the software-engineering community discourage the last option. See Scott Meyers, "Item 33: Make non-leaf classes abstract," *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1995. Also see, Herb Sutter, "Virtuality," *C/C++ Users Journal*, vol. 19, no. 9, September 2001. <http://www.gotw.ca/publications/mill18.htm>.

Let's develop a simple inheritance hierarchy, represented by the following **UML class diagram**—such diagrams illustrate the relationships among classes in a hierarchy:



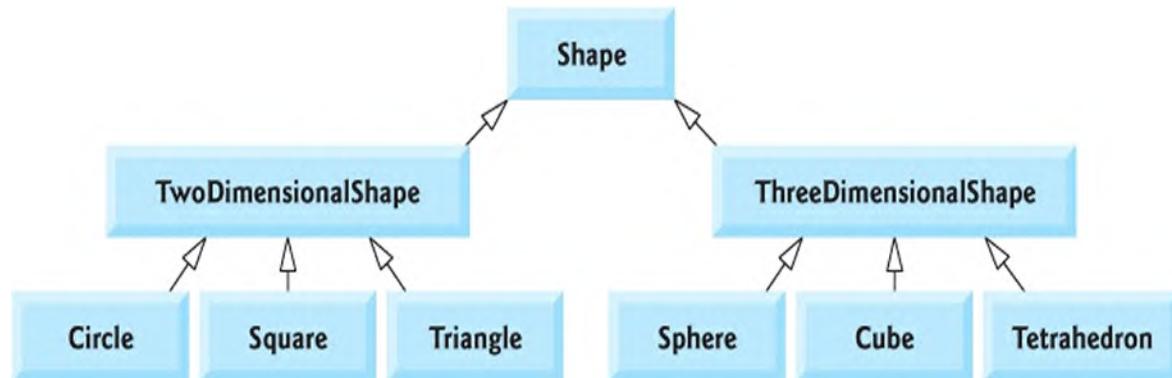
A university community might have thousands of `CommunityMembers`, such as `Employees`, `Students` and `alumni` (each of class `Alumnus`). `Employees` are either `Faculty` or `Staff`, and `Faculty` are either `Administrators` or `Teachers`. Some `Administrators` are also `Teachers`. With **single inheritance**, a derived class inherits from *one* base class. With **multiple inheritance**, a derived class inherits from two or more base classes. In this hierarchy, we've used multiple inheritance to form class `AdministratorTeacher`.

Each upward-pointing arrow in the diagram represents an *is-a* relationship. As we follow the arrows upward, we can state “*an Employee is a CommunityMember*” and “*a Teacher is a Faculty member*.” `CommunityMember` is the **direct base class** of `Employee`, `Student` and `Alumnus`. `CommunityMember` is an **indirect base class** of all the hierarchy's other classes. An indirect base class is two or more levels up the class hierarchy from its derived classes.

You can follow the arrows upward several levels, applying the *is-a* relationship. So, an `AdministratorTeacher` *is an* `Administrator` (and *is also a* `Teacher`), *is a* `Faculty` member, *is an* `Employee` and *is a* `CommunityMember`.

10.2.2 Shape Class Hierarchy and public Inheritance

Let's consider a `Shape` inheritance hierarchy that begins with base-class `Shape`:



Classes `TwoDimensionalShape` and `ThreeDimensionalShape` derive from `Shape`, so a `TwoDimensionalShape` *is a* `Shape`, and a `ThreeDimensionalShape` *is a* `Shape`. The hierarchy's third level contains specific `TwoDimensionalShapes` and

`ThreeDimensionalShapes`. We can follow the arrows upward to identify direct and indirect *is-a* relationships. For instance, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`.

The following class header specifies that `TwoDimensionalShape` inherits from `Shape`:

[Click here to view code image](#)

```
class TwoDimensionalShape : public Shape
```

This is **public inheritance**, which we'll use in this chapter. With public inheritance, public base-class members become public derived-class members and, as we'll discuss later in this chapter, protected base-class members become protected derived-class members. private base-class members are not directly accessible in that class's derived classes. However, these members are still inherited and part of the derived-class objects. The derived class can manipulate private base-class members through inherited base-class public and protected member functions—if these base-class member functions provide such functionality. We'll also discuss private and protected inheritance ([Section 10.16](#)).

 **Inheritance is not appropriate for every class relationship.**

Composition's *has-a* relationship sometimes is more appropriate. For example, given `Employee`, `BirthDate` and `PhoneNumber` classes, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `PhoneNumber`. However, an `Employee` *has a* `BirthDate` and *has a* `PhoneNumber`.

It's possible to treat base-class objects and derived-class objects similarly —their commonalities are expressed in the base-class members. Later in this chapter, we'll consider examples that take advantage of that relationship.

10.3 Relationship between Base and Derived Classes

This section uses an inheritance hierarchy of employee types in a company's payroll application to demonstrate the relationship between base and derived classes:

- Base-class salaried employees are paid a fixed weekly salary.
- Derived-class salaried commission employees receive a weekly salary *plus* a percentage of their sales.

10.3.1 Creating and Using a `SalariedEmployee` Class

Let's examine `SalariedEmployee`'s class definition (Figs. 10.1–10.2). Its header ([Fig. 10.1](#)) specifies the class's public services:

- a constructor (line 9),
- member function `earnings` (line 17),
- member function `toString` (line 18), and
- public *get* and *set* functions that manipulate the class's data members `m_name` and `m_salary` (declared in lines 20–21).

Member function `setSalary`'s implementation ([Fig. 10.2](#), lines 23–29) validates its argument before modifying the data member `m_salary`.

[Click here to view code image](#)

```

1 // Fig. 10.1: SalariedEmployee.h
2 // SalariedEmployee class definition.
3 #pragma once // prevent multiple inclusions
4 #include <string>
5 #include <string_view>
6
7 class SalariedEmployee {
8 public:
9     SalariedEmployee(std::string_view name, .
10
11     void setName(std::string_view name);
12     std::string getName() const;
13
14     void setSalary(double salary);
15     double getSalary() const;
16
17     double earnings() const;

```

```
18     std::string toString() const;
19 private:
20     std::string m_name{};
21     double m_salary{0.0};
22 }
```



Fig. 10.1 | SalariedEmployee class definition.

[Click here to view code image](#)

```
1 // Fig. 10.2: SalariedEmployee.cpp
2 // Class SalariedEmployee member-function definitions
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be <format>
5 #include "SalariedEmployee.h" // SalariedEmployee.h must be included
6 using namespace std;
7
8 // constructor
9 SalariedEmployee::SalariedEmployee(string_view name,
10 : m_name{name} {
11     setSalary(salary);
12 }
13
14 // set name
15 void SalariedEmployee::setName(string_view name) {
16     m_name = name; // should validate
17 }
18
19 // return name
20 string SalariedEmployee::getName() const { return m_name; }
21
22 // set salary
23 void SalariedEmployee::setSalary(double salary) {
24     if (salary < 0.0) {
25         throw invalid_argument("Salary must be non-negative");
26     }
27 }
```

```
26     }
27
28     m_salary = salary;
29 }
30
31 // return salary
32 double SalariedEmployee::getSalary() const
33
34 // calculate earnings
35 double SalariedEmployee::earnings() const {
36
37 // return string representation of Salaried
38 string SalariedEmployee::toString() const {
39     return fmt::format("name: {}\nsalary: ${{
40             getSalary()});
41 }
```



Fig. 10.2 | Class `SalariedEmployee` member-function definitions.

SalariedEmployee Constructor

The class's constructor (Fig. 10.2, lines 9–12) uses a member initializer list to initialize `m_name`. We could validate the name, perhaps by ensuring that it's of a reasonable length. The constructor calls `setSalary` to validate and initialize data member `m_salary`.

SalariedEmployee Member Functions `earnings` and `toString`

Function `earnings` (line 35) calls `getSalary` and returns the result, and `toString` (lines 38–41) returns a string containing a `SalariedEmployee` object's data-member values.

Testing Class SalariedEmployee

Figure 10.3 tests class `SalariedEmployee`. Line 10 creates the `SalariedEmployee` object `employee`. Lines 13–15 demonstrate the employee's *get* functions. Line 17 uses `setSalary` to change the employee's `m_salary` value. Then, lines 18–19 call `employee`'s

`toString` member function to get and output the employee's updated information. Finally, line 22 displays the employee's earnings, using the updated `m_salary` value.

[Click here to view code image](#)

```
1 // fig10_03.cpp
2 // SalariedEmployee class test program.
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this w
5 #include "SalariedEmployee.h" // SalariedEmpl
6 using namespace std;
7
8 int main() {
9     // instantiate a SalariedEmployee object
10    SalariedEmployee employee{"Sue Jones", 300.0};
11
12    // get SalariedEmployee data
13    cout << "Employee information obtained by
14        << fmt::format("name: {}\\nsalary: ${}
15                    employee.getSalary());
16
17    employee.setSalary(500.0); // change salary
18    cout << "\\nUpdated employee information
19        << employee.toString();
20
21    // display only the employee's earnings
22    cout << fmt::format("\\nearnings: ${:.2f}")
23 }
```

< >

```
Employee information obtained by get functions:
name: Sue Jones
salary: $300.00
```

```
Updated employee information from function toString
```

```
name: Sue Jones  
salary: $500.00  
  
earnings: $500.00
```

Fig. 10.3 | SalariedEmployee class test program.

10.3.2 Creating a `SalariedEmployee` – `SalariedCommissionEmployee` Inheritance Hierarchy

Let's create a `SalariedCommissionEmployee` class (Figs. 10.4–10.5) that inherits from `SalariedEmployee` (Figs. 10.1–10.2). In this example, a `SalariedCommissionEmployee` object *is a* `SalariedEmployee` —public inheritance passes on `SalariedEmployee`'s capabilities. A `SalariedCommissionEmployee` also has `m_grossSales` and `m_commissionRate` data members (Fig. 10.4, lines 22–23), which we'll multiply to calculate the commission earned. Lines 13–17 declare public *get* and *set* functions that manipulate the class's data members.

[Click here to view code image](#)

```
1 // Fig. 10.4: SalariedCommissionEmployee.h
2 // SalariedCommissionEmployee class derived
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "SalariedEmployee.h"
7
8 class SalariedCommissionEmployee : public SalariedEmployee
9 {
10     SalariedCommissionEmployee(std::string_view name,
11                               double grossSales, double commissionRate);
12
13     void setGrossSales(double grossSales);
14     double getGrossSales() const;
```

```

15
16     void setCommissionRate(double commission);
17     double getCommissionRate() const;
18
19     double earnings() const;
20     std::string toString() const;
21 private:
22     double m_grossSales{0.0};
23     double m_commissionRate{0.0};
24 };

```



Fig. 10.4 | SalariedCommissionEmployee class definition indicating inheritance relationship with class CommissionEmployee.

Inheritance

SE A The **colon (:) in line 8 indicates inheritance**. The keyword `public` specifies the inheritance type. With `public` inheritance, `public` base-class members remain `public` in the derived class and, as you'll see later, `protected` base-class members remain `protected` in the derived class. `SalariedCommissionEmployee` inherits all of `SalariedEmployee`'s members, *except* its constructor(s) and destructor. **Constructors and destructors are specific to the class that defines them, so derived classes have their own.⁵**

5. It's common in a derived-class constructor to simply pass its arguments to a corresponding base-class constructor and do nothing else. We'll see in Chapter 19, Other Topics, that inserting a `using` declaration in a derived class enables you to inherit the base class's constructors.

SalariedCommissionEmployee Member Functions

Class `SalariedCommissionEmployee`'s public services (Fig. 10.4) include:

- its own constructor (lines 10–11),
- the member functions `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `toString` (lines 13–20), and

- the public member functions inherited from class SalariedEmployee.

SE A Although `SalariedCommissionEmployee`'s source code does not contain these inherited members, they're nevertheless a part of the class. A `SalariedCommissionEmployee` object also contains `SalariedEmployee`'s private members, but **they're not directly accessible** within the derived class. You can access them only via the inherited `SalariedEmployee` public (or protected) member functions.

`SalariedCommissionEmployee`'s Implementation

SE A Figure 10.5 shows `SalariedCommissionEmployee`'s member-function implementations. **Each derived-class constructor must call a base-class constructor.** We do this explicitly via a **base-class initializer** (line 11)—that is, a member initializer that passes arguments to a base-class constructor. In the `SalariedCommissionEmployee` constructor (lines 9–15), line 11 calls `SalariedEmployee`'s constructor, which initializes the inherited data members with the arguments `name` and `salary`. A base-class initializer explicitly invokes the base-class constructor by name. Member functions `setGrossSales` (lines 18–24) and `setCommissionRate` (lines 32–40) validate their arguments before modifying the data members `m_grossSales` and `m_commissionRate`.

[Click here to view code image](#)

```

1 // Fig. 10.5: SalariedCommissionEmployee.cpp
2 // Class SalariedCommissionEmployee member-
3 #include "fmt/format.h" // In C++20, this will
4 #include <stdexcept>
5 #include "SalariedCommissionEmployee.h"
6 using namespace std;
7
8 // constructor
9 SalariedCommissionEmployee::SalariedCommissionEmployee(
10     double salary, double grossSales, double commissionRate)
11 : SalariedEmployee(name, salary) {
12     m_grossSales = grossSales;
13     m_commissionRate = commissionRate;
14 }
15
16 void SalariedCommissionEmployee::setGrossSales(double grossSales) {
17     if (grossSales < 0)
18         throw invalid_argument("grossSales must be >= 0");
19     m_grossSales = grossSales;
20 }
21
22 void SalariedCommissionEmployee::setCommissionRate(double commissionRate) {
23     if (commissionRate < 0)
24         throw invalid_argument("commissionRate must be >= 0");
25     m_commissionRate = commissionRate;
26 }
```

```
11     : SalariedEmployee{name, salary} { // ca
12
13     setGrossSales(grossSales); // validate &
14     setCommissionRate(commissionRate); // va
15 }
16
17 // set gross sales amount
18 void SalariedCommissionEmployee::setGrossSa
19     if (grossSales < 0.0) {
20         throw invalid_argument("Gross sales m
21     }
22
23     m_grossSales = grossSales;
24 }
25
26 // return gross sales amount
27 double SalariedCommissionEmployee::getGross
28     return m_grossSales;
29 }
30
31 // set commission rate
32 void SalariedCommissionEmployee::setCommiss
33     double commissionRate) {
34
35     if (commissionRate <= 0.0 || commissionR
36         throw invalid_argument("Commission ra
37     }
38
39     m_commissionRate = commissionRate;
40 }
41
42 // get commission rate
43 double SalariedCommissionEmployee::getCommi
44     return m_commissionRate;
45 }
46
47 // calculate earnings--uses SalariedEmploye
```

```

48     double SalariedCommissionEmployee::earnings
49         return SalariedEmployee::earnings() +
50             getGrossSales() * getCommissionRate()
51     }
52
53 // returns string representation of SalariedCommissionEmployee
54 string SalariedCommissionEmployee::toString
55     return fmt::format(
56         "{}gross sales: ${:.2f}\ncommission rate: {:.2f}",
57         SalariedEmployee::toString(), getGrossSales(),
58     )

```



Fig. 10.5 | Class `SalariedCommissionEmployee` member-function definitions. (Part 2 of 2.)

SalariedCommissionEmployee Member Function `earnings`

`SalariedCommissionEmployee`'s `earnings` function (lines 48–51) redefines `earnings` from class `SalariedEmployee` (Fig. 10.2, line 35) to calculate a `SalariedCommissionEmployee`'s earnings. Line 49 in `SalariedCommissionEmployee`'s version uses the expression

[Click here to view code image](#)

`SalariedEmployee::earnings()`

to get the portion of the earnings based on salary, then adds that value to the commission to calculate the total earnings.

SE A Err X To call a redefined base-class member function from the derived class, place the base-class name and the scope resolution operator (`::`) before the base-class member-function name. `SalariedEmployee::` is required here to avoid infinite recursion. Also, by calling `SalariedEmployee`'s `earnings` function from `SalariedCommissionEmployee`'s `earnings` function, we avoid duplicating code.⁶

6. That's the upside. The downside is that this creates a coupling between base and derived classes that can make large-scale systems difficult to modify.

SalariedCommissionEmployee Member Function `toString`

Similarly, `SalariedCommissionEmployee`'s `toString` function (Fig. 10.5, lines 54–58) **redefines** class `SalariedEmployee`'s `toString` function (Fig. 10.2, lines 38–41). The new version returns a string containing:

- the result of calling `SalariedEmployee::toString()` (Fig. 10.5, line 57), and
- the `SalariedCommissionEmployee`'s gross sales and commission rate.

Testing Class `SalariedCommissionEmployee`

Figure 10.6 creates the `SalariedCommissionEmployee` object `employee` (line 10). Lines 13–17 output the `employee`'s data by calling its *get* functions to retrieve the object's data member values. Lines 19–20 use `setGrossSales` and `setCommissionRate` to change the `employee`'s `m_grossSales` and `m_commissionRate` values, respectively. Then, lines 21–22 call `employee`'s `toString` member function to get and output the `employee`'s updated information. Finally, line 25 displays the `employee`'s earnings, which are calculated using the updated `m_grossSales` and `m_commissionRate` values.

[Click here to view code image](#)

```
1 // fig10_06.cpp
2 // SalariedCommissionEmployee class test pr...
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this w...
5 #include "SalariedCommissionEmployee.h"
6 using namespace std;
7
8 int main() {
9     // instantiate SalariedCommissionEmployee
10    SalariedCommissionEmployee employee{"Bob",
11
12        // get SalariedCommissionEmployee data
```

```

13     cout << "Employee information obtained by
14         << fmt::format("{}: {}\\n{}: {:.2f}\\n",
15             "name", employee.getName(), "
16             "gross sales", employee.getGrossSales(),
17             "commission", employee.getCommissionRate());
18
19     employee.setGrossSales(8000.0); // change
20     employee.setCommissionRate(0.1); // change
21     cout << "\\nUpdated employee information
22         << employee.toString();
23
24     // display the employee's earnings
25     cout << fmt::format("\\nearnings: ${:.2f}\n");
26 }
```

< >

```

Employee information obtained by get functions:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission: 0.04
```

Updated employee information from function toString()
name: Bob Lewis
salary: \$300.00
gross sales: \$8000.00
commission rate: 0.10

earnings: \$1100.00

< >

Fig. 10.6 | SalariedCommissionEmployee class test program.

A Derived-Class Constructor Must Call Its Base Class's Constructor

The compiler would issue an error if SalariedCommissionEmployee's constructor did not invoke class SalariedEmployee's constructor

explicitly. In this case, the compiler would attempt to call class SalariedEmployee's default constructor, which does not exist because the base class explicitly defined a constructor. **If the base-class provides a default constructor, the derived-class constructor can call the base-class constructor implicitly.**

Notes on Constructors in Derived Classes

 **Err** A derived-class constructor must call its base class's constructor with any required arguments; otherwise, a compilation error occurs. This ensures that inherited **private** base-class members, which the derived class cannot access directly, get initialized. The derived class's data-member initializers are typically placed after base-class initializers in the member-initializer list.

Base-Class **private** Members Are Not Directly Accessible in a Derived Class

 C++ rigidly enforces restrictions on accessing private data members. Even a derived class, which is intimately related to its base class, cannot directly access its base class's **private** data. For example, class SalariedEmployee's private `m_salary` data member, though part of each SalariedCommissionEmployee object, cannot be accessed by class SalariedCommissionEmployee's member functions directly. If they try, the compiler produces an error message, such as the following from GNU g++:

[Click here to view code image](#)

```
'double SalariedEmployee::m_salary' is private wi
```



However, as you saw in Fig. 10.5, SalariedCommissionEmployee's member functions can access the public members inherited from class SalariedEmployee.

Including the Base-Class Header in the Derived-Class Header

Notice that we `#include` the base class's header in the derived class's header (Fig. 10.4, line 6). This is necessary for several reasons:

- For the derived class to inherit from the base class in line 8 (Fig. 10.4), the compiler requires the base-class definition from SalariedEmployee.h.
 - **SE A** **The compiler determines an object's size from its class's definition.** A program that creates an object must know the class definition so the compiler can reserve the proper amount of memory for the object. **A derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes.**⁷ Including the base class's definition allows the compiler to determine the complete memory requirements for all the data members that contribute to a derived-class object's total size.
7. All objects of a class share one copy of the class's member functions, which are stored separately from those objects and are not part of their size.
- The base-class definition also enables the compiler to determine whether the derived class uses the base class's inherited members properly. For example, **the compiler prevents a derived class from accessing the base class's private data directly. The compiler also uses the base class's function prototypes to validate derived-class function calls to inherited base-class functions.**

Eliminating Repeated Code Via Implementation Inheritance

SE A With implementation inheritance, the base class declares the common data members and member functions of all the classes in the hierarchy. When changes are required for these common features, you make the changes only in the base class. The derived classes then inherit the changes and must be recompiled. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.⁸

8. Again, the downside of implementation inheritance, especially in deep inheritance hierarchies, is that it creates a tight coupling among the classes in the inheritance hierarchy, making it difficult to modify.

10.4 Constructors and Destructors in Derived Classes

Order of Constructor Calls

SE A Instantiating a derived-class object begins a **chain of constructor calls**. **Before performing its own tasks, a derived-class constructor invokes its direct base class's constructor either explicitly via a base-class member initializer or implicitly by calling the base class's default constructor.** The last constructor called in this chain is for the base class at the top of the hierarchy. That constructor finishes executing *first*, and the most-derived-class constructor's body finishes executing *last*.

Each base-class constructor initializes the base-class data members that its derived-classes inherit. In the SalariedEmployee/SalariedCommissionEmployee hierarchy we've been studying, when we create a SalariedCommissionEmployee object:

- Its constructor immediately calls the SalariedEmployee constructor.
- SalariedEmployee is this hierarchy's base class, so the SalariedEmployee constructor executes, initializing the SalariedCommissionEmployee object's inherited SalariedEmployee `m_name` and `m_salary` data members.
- SalariedEmployee's constructor then returns control to SalariedCommissionEmployee's constructor to initialize `m_grossSales` and `m_commissionRate`.

Order of Destructor Calls

SE A When a derived-class object is destroyed, the program calls that object's destructor. This begins a **chain of destructor calls**. **The destructors execute in the reverse order of the constructors.** When a derived-class object's destructor is called, it performs its task, then invokes the destructor of the next class up the hierarchy. This repeats until the destructor of the base class at the hierarchy's top is called.

Constructors and Destructors for Composed Objects

SE A Suppose we create a derived-class object where both the base class and the derived class are composed of objects of other classes. When a

derived-class object is created:

- First, the constructors for the base class's member objects execute in the order those objects were declared,
- then the base-class constructor body executes,
- then the constructors for the derived class's member objects execute in the order those objects were declared in the derived class,
- then the derived class's constructor body executes.

Destructors for data members are called in the reverse of the order from which their corresponding constructors were called.

10.5 Intro to Runtime Polymorphism: Polymorphic Video Game

Suppose we're designing a video game containing Martians, Venusians, Plutonians, SpaceShips and LaserBeams. Each inherits from a SpaceObject base class with the member function draw and implements this function in a manner appropriate to the derived class.

Screen Manager

A screen-manager program maintains a vector of SpaceObject pointers to objects of the various classes. **To refresh the screen, the screen manager periodically sends each object the same draw message, and each object responds in its unique way.** For example,

- a Martian might draw itself in red with the appropriate number of antennae,
- a SpaceShip might draw itself as a silver flying saucer, and
- a LaserBeam might draw itself as a bright red beam across the screen.

The same draw message has **many forms** of results—hence the term **polymorphism**.

Adding New Classes to the System

A polymorphic screen manager facilitates adding new classes to a system with minimal code modifications. Suppose we want to add Mercurian

objects to our game. We build a class `Mercurian` that inherits from `SpaceObject` and defines `draw`, then add the object's address to the vector of `SpaceObject` pointers. The screen-manager code invokes member function `draw` the same way for every object the vector points to, regardless of the object's type. So, the new `Mercurian` objects just “**plug right in.**” **Without modifying the system—other than to create objects of the new classes—you can use runtime polymorphism to accommodate additional classes, including ones not envisioned when the system was created.**

SE A Runtime polymorphism enables you to deal in **generalities** and let the execution-time environment concern itself with the **specifics**. You can direct objects to behave in appropriate manners without knowing their types, as long as those objects belong to the same class hierarchy and are being accessed via a common base-class pointer or reference.⁹

9. We'll see that there are forms of runtime polymorphism and compile-time polymorphism that do not depend on inheritance hierarchies.

SE A Runtime polymorphism promotes **extensibility**. Software that invokes polymorphic behavior is written independently of the specific object types to which messages are sent. Thus, new object types that can respond to existing messages can simply be plugged into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

10.6 Relationships Among Objects in an Inheritance Hierarchy

Section 10.3 created a
SalariedEmployee–SalariedCommissionEmployee class hierarchy. Let's examine the relationships among classes in an inheritance hierarchy more closely.

The next several sections present examples demonstrating how base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects:

- In [Section 10.6.1](#), we assign a derived-class object's address to a base-class pointer, then show that invoking a function via the base-class pointer invokes the *base-class* functionality in the derived-class object. **The handle's type determines which function is called.**
- In [Section 10.6.2](#), we assign a base-class object's address to a derived-class pointer to show the resulting compilation error. We discuss the error message and investigate why the compiler does not allow such an assignment.
- In [Section 10.6.3](#), we assign a derived-class object's address to a base-class pointer, then examine how the base-class pointer can be used to invoke only the base-class functionality. **When we attempt to invoke derived-class-only member functions through the base-class pointer, compilation errors occur.**
- Finally, in [Section 10.7](#), we introduce **virtual functions** to demonstrate how to get **runtime polymorphic behavior** from base-class pointers aimed at derived-class objects. We then **assign a derived-class object's address to a base-class pointer and use that pointer to invoke derived-class functionality—precisely what we need to achieve runtime polymorphic behavior.**

These examples demonstrate that **with public inheritance, an object of a derived class can be treated as an object of its base class**. This enables various interesting manipulations. For example, a program can create a vector of base-class pointers that point to objects of many derived-class types. The compiler allows this because each derived-class object *is an object of its base class*.

SE A On the other hand, **you cannot treat a base-class object as an object of a derived class**. For example, a `SalariedEmployee` is not a `SalariedCommissionEmployee`—it's missing the data members `m_grossSales` or `m_commissionRate` and does not have their corresponding *set* and *get* member functions. **The is-a relationship applies only from a derived class to its direct and indirect base classes.**

10.6.1 Invoking Base-Class Functions from Derived-Class Objects

Figure 10.7 reuses classes `SalariedEmployee` and `SalariedCommissionEmployee` from [Sections 10.3.1–10.3.2](#). The example demonstrates aiming base-class and derived-class pointers at base-class and derived-class objects. The first two are natural and straightforward:

- we aim a base-class pointer at a base-class object and invoke base-class functionality, and
 - we aim a derived-class pointer at a derived-class object and invoke derived-class functionality.

Then, we demonstrate the *is-a* relationship between derived classes and base classes by aiming a base-class pointer at a derived-class object and showing that **the base-class functionality is indeed available in the derived-class object**.

[Click here to view code image](#)

```
1 // fig10_07.cpp
2 // Aiming base-class and derived-class pointer
3 // and derived-class objects, respectively.
4 #include <iostream>
5 #include "fmt/format.h" // In C++20, this will
6 #include "SalariedEmployee.h"
7 #include "SalariedCommissionEmployee.h"
8 using namespace std;
9
10 int main() {
11     // create base-class object
12     SalariedEmployee salaried{"Sue Jones", 5000.0};
13
14     // create derived-class object
15     SalariedCommissionEmployee salariedCommissioner{"Bob Lewis", 300.0, 5000.0, .04};
16
17     // output objects salaried and salariedCommissioner
18     cout << fmt::format("{}:\n{}:\n{}\n", salaried, salariedCommissioner);
19
20 }
```

```

21         salaried.toString(), // base-
22         salariedCommission.toString()
23
24     // natural: aim base-class pointer at ba-
25     SalariedEmployee* salariedPtr{&salaried}
26     cout << fmt::format("{}\n{}:\n{}\n",
27                         "CALLING TOSTRING WITH BASE-C-
28                         "BASE-CLASS OBJECT INVOKES BA-
29                         salariedPtr->toString()); // :
30
31     // natural: aim derived-class pointer at
32     SalariedCommissionEmployee* salariedComm-
33
34     cout << fmt::format("{}\n{}:\n{}\n",
35                         "CALLING TOSTRING WITH DERIVE-
36                         "DERIVED-CLASS OBJECT INVOKES
37                         salariedCommissionPtr->toStri-
38
39     // aim base-class pointer at derived-cla-
40     salariedPtr = &salariedCommission;
41     cout << fmt::format("{}\n{}:\n{}\n",
42                         "CALLING TOSTRING WITH BASE-C-
43                         "OBJECT INVOKES BASE-CLASS FU-
44                         salariedPtr->toString()); // :
45 }
```

< >

DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:

name: Sue Jones

salary: \$500.00

name: Bob Lewis

salary: \$300.00

gross sales: \$5000.00

commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO

```
BASE-CLASS OBJECT INVOOKES BASE-CLASS FUNCTIONALITY:  
name: Sue Jones  
salary: $500.00  
  
CALLING TOSTRING WITH DERIVED-CLASS POINTER TO  
DERIVED-CLASS OBJECT INVOOKES DERIVED-CLASS FUNCT  
name: Bob Lewis  
salary: $300.00  
gross sales: $5000.00  
commission rate: 0.04  
  
CALLING TOSTRING WITH BASE-CLASS POINTER TO DERI  
OBJECT INVOOKES BASE-CLASS FUNCTIONALITY:  
name: Bob Lewis  
salary: $300.00
```

Fig. 10.7 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 2.)

Recall that a `SalariedCommissionEmployee` object is a `SalariedEmployee` that also earns a commission based on gross sales. `SalariedCommissionEmployee`'s `earnings` member function (Fig. 10.5, lines 48–51) **redefines** `SalariedEmployee`'s version (Fig. 10.2, line 35) to include the commission. `SalariedCommissionEmployee`'s `toString` member function (Fig. 10.5, lines 54–58) **redefines** `SalariedEmployee`'s version (Fig. 10.2, lines 38–41) to return the same information and the employee's commission.

Creating Objects and Displaying Their Contents

In Fig. 10.7, line 12 creates a `SalariedEmployee` object and lines 15–16 create a `SalariedCommissionEmployee` object. Lines 21 and 22 use each object's name to invoke its `toString` member function.

Aiming a Base-Class Pointer at a Base-Class Object

Line 25 initializes the `SalariedEmployee` pointer `salariedPtr` with the base-class object `salaried`'s address. Line 29 uses the pointer to

invoke the `salaried` object's `toString` member function from base-class `SalariedEmployee`.

Aiming a Derived-Class Pointer at a Derived-Class Object

Line 32 initializes `SalariedCommissionEmployee` pointer `salariedCommissionPtr` with derived-class object `salariedCommission`'s address. Line 37 uses the pointer to invoke the `salariedCommission` object's `toString` member function from derived-class `SalariedCommissionEmployee`.

Aiming a Base-Class Pointer at a Derived-Class Object

Line 40 assigns derived-class object `salariedCommission`'s address to `salariedPtr`—a base-class pointer. **This “crossover” is allowed because a derived-class object is an object of its base class.** Line 44 uses this pointer to invoke member function `toString`. **Even though the base-class `SalariedEmployee` pointer points to a `SalariedCommissionEmployee` derived-class object, the base class's `toString` member function is invoked.** The gross sales and the commission rate are not displayed because they are not base-class members.

This program's output shows that **the invoked function depends on the pointer type (or reference type, as you'll see) used to invoke the function, not the object type for which the member function is called.** In [Section 10.7](#), we'll see that **with virtual functions, it's possible to invoke the object type's functionality rather than that of the pointer (or reference) type**—an important aspect of implementing runtime polymorphic behavior.

10.6.2 Aiming Derived-Class Pointers at Base-Class Objects

Now, let's try to aim a derived-class pointer at a base-class object ([Fig. 10.8](#)). Line 7 creates a `SalariedEmployee` object. Line 11 attempts to initialize a `SalariedCommissionEmployee` pointer with the base-class `salaried` object's address. The compiler generates an error because a `SalariedEmployee` is *not* a `SalariedCommissionEmployee`.

[Click here to view code image](#)

```
1 // fig10_08.cpp
2 // Aiming a derived-class pointer at a base
3 #include "SalariedEmployee.h"
4 #include "SalariedCommissionEmployee.h"
5
6 int main() {
7     SalariedEmployee salaried{"Sue Jones", 5
8
9     // aim derived-class pointer at base-cla
10    // Error: a SalariedEmployee is not a Sa
11    SalariedCommissionEmployee* salariedComm
12 }
```



Microsoft Visual C++ compiler error message:

```
fig10_08.cpp(11,63): error C2440: 'initializing'
'SalariedEmployee *' to 'SalariedCommissionEmplo'
```

Fig. 10.8 | Aiming a derived-class pointer at a base-class object.

SE  Consider the consequences if the compiler were to allow this assignment. Through a `SalariedCommissionEmployee` pointer, we can invoke any of that class's member functions, including `setGrossSales` and `setCommissionRate`, for the object to which the pointer points—the base-class object `salaried`. However, a `SalariedEmployee` object has neither `setGrossSales` and `setCommissionRate` member functions nor data members `m_grossSales` and `m_commissionRate` to set. **This could lead to problems because member functions `setGrossSales` and `setCommissionRate` would assume data members `m_grossSales` and `m_commissionRate` exist at their “usual locations” in a `SalariedCommissionEmployee` object.** A `SalariedEmployee` object's memory does not have these data members, so

setGrossSales and **setCommissionRate** might overwrite other data in memory, possibly data belonging to a different object.

10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers

The compiler allows us to invoke only base-class member functions via a base-class pointer. So, if a base-class pointer is aimed at a derived-class object, and an attempt is made to access a derived-class-only member function, a compilation error occurs. Figure 10.9 shows the compiler errors for invoking a derived-class-only member function via a base-class pointer (lines 23–25).

[Click here to view code image](#)

```
1 // fig10_09.cpp
2 // Attempting to call derived-class-only fu:
3 // via a base-class pointer.
4 #include <string>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7 using namespace std;
8
9 int main() {
10     SalariedCommissionEmployee salariedComm
11         "Bob Lewis", 300.0, 5000.0, .04};
12
13 // aim base-class pointer at derived-cla
14 SalariedEmployee* salariedPtr{&salariedC
15
16 // invoke base-class member functions on
17 // object through base-class pointer (al
18 string name{salariedPtr->getName()};
19 double salary{salariedPtr->getSalary()};
20
21 // attempt to invoke derived-class-only :
22 // on derived-class object through base-
```

```
23     double grossSales{salariedPtr->getGrossS
24     double commissionRate{salariedPtr->getCo
25     salariedPtr->setGrossSales(8000.0);
26 }
```



GNU C++ compiler error messages:

```
fig10_09.cpp: In function 'int main()':
fig10_09.cpp:23:35: error: 'class SalariedEmploy
'getGrossSales'
    23 |     double grossSales{salariedPtr->getGr
          |                         ^
          |                         ~~~~~
fig10_09.cpp:24:39: error: 'class SalariedEmploy
'getCommissionRate'
    24 |     double commissionRate{salariedPtr->g
          |                         ^
          |                         ^
fig10_09.cpp:25:17: error: 'class SalariedEmploy
'setGrossSales'
    25 |     salariedPtr->setGrossSales(8000.0);
          |                         ^
          |                         ~~~~~~
```



Fig. 10.9 | Attempting to call derived-class-only functions via a base-class pointer.

Lines 10–11 create a `SalariedCommissionEmployee` object. Line 14 initializes the base-class pointer `salariedPtr` with the derived-class object `salariedCommission`'s address. Again, this is allowed because a `SalariedCommissionEmployee` is a `SalariedEmployee`.

Lines 18–19 invoke base-class member functions via the base-class pointer. These calls are allowed because `SalariedCommissionEmployee` inherits these functions.

We know that `salariedPtr` is aimed at a `SalariedCommissionEmployee` object, so lines 23–25 try to invoke `SalariedCommissionEmployee`-only member functions

`getGrossSales`, `getCommissionRate` and `setGrossSales`. The compiler generates errors because these functions are not base-class `SalariedEmployee` member functions. Through `salariedPtr` we can invoke only those functions that are members of the handle's class type—in this case, only base-class `SalariedEmployee` member functions.

10.7 Virtual Functions and Virtual Destructors

In Section 10.6.1, we aimed a base-class `SalariedEmployee` pointer at a derived-class `SalariedCommissionEmployee` object, then used it to invoke member function `toString`. In that case, the `SalariedEmployee` class's `toString` was called. How can we invoke the derived-class `toString` function via a base-class pointer?

10.7.1 Why `virtual` Functions Are Useful

SE A Suppose the shape classes `Circle`, `Triangle`, `Rectangle` and `Square` all derive from base-class `Shape`. Each class might be endowed with the ability to draw objects of that class via a member function `draw`, but each shape's implementation is quite different. In a program that draws many different types of shapes, it would be convenient to treat them all generally as base-class `Shape` objects. We could then draw any shape by using a base-class `Shape` pointer to invoke function `draw`. The program would determine dynamically at runtime which derived-class `draw` function to use, based on the type of the object to which the base-class `Shape` pointer points. This is runtime polymorphic behavior. **With `virtual` functions, the type of the object pointed to (or referenced)—not the type of the pointer (or reference) handle—determines which member function to invoke.**

10.7.2 Declaring `virtual` Functions

SE A To enable this runtime polymorphic behavior, we declare the base-class member function `draw` as `virtual`¹⁰, then `override` it in each derived class to draw the appropriate shape. **An overridden function in a derived class must have the same signature as the base-class function it overrides.** To declare a `virtual` function, precede its prototype with the keyword `virtual`. For example,

10. Some programming languages, such as Java and Python, treat all member functions (methods) like C++ virtual functions. As we'll see in [Section 10.10](#), virtual functions have a slight execution-time performance hit and a slight memory consumption hit. C++ allows you to choose whether to make each function virtual or not, based on the performance requirements of your applications.

```
virtual void draw() const;
```

would appear in base-class `Shape`. The preceding prototype declares that function `draw` is a virtual function that takes no arguments and returns nothing. This function is declared `const` because a `draw` function should not modify the `Shape` object on which it's invoked.

 Virtual functions do not have to be `const` and can receive arguments and return values as appropriate. **Once a function is declared `virtual`, it's `virtual` in all classes derived directly or indirectly from that base class.** If a derived class does not override a virtual function from its base class, the derived class simply inherits its base class's virtual function implementation.

10.7.3 Invoking a `virtual` Function

If a program invokes a virtual function through

- a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`) or
- a base-class reference to a derived-class object (e.g., `shapeRef.draw()`),

the program will choose the correct *derived-class* function at execution time, based on the object's type—not the pointer or reference type. Choosing the appropriate function to call at execution time is known as **dynamic binding** or **late binding**.

When a virtual function is called by referring to a specific object *by name* and using the dot operator (e.g., `squareObject.draw()`), an optimizing compiler can resolve at *compile time* the function to call. This is called **static binding**. The virtual function that will be called is the one defined for that object's class.

10.7.4 `virtual` Functions in the `SalariedEmployee`

Hierarchy

Let's see how virtual functions could enable runtime polymorphic behavior in our hierarchy. To enable the behavior you'll see in Fig. 10.10, we made only two modifications to each class's header. In class SalariedEmployee's header (Fig. 10.1), we modified the earnings and `toString` prototypes (lines 17–18)

[Click here to view code image](#)

```
double earnings() const;  
std::string toString() const;
```

to **include the `virtual` keyword**, as in:

[Click here to view code image](#)

```
virtual double earnings() const;  
virtual std::string toString() const;
```

In class SalariedCommissionEmployee's header (Fig. 10.4), we modified the earnings and `toString` prototypes (lines 19–20)

[Click here to view code image](#)

```
double earnings() const;  
std::string toString() const;
```

to **include the `override` keyword** (discussed after Fig. 10.10), as in:

[Click here to view code image](#)

```
double earnings() const override;  
std::string toString() const override;
```

SalariedEmployee's `earnings` and `toString` functions are `virtual`, so derived-class SalariedCommissionEmployee's versions *override* SalariedEmployee's versions. There were no changes to the SalariedEmployee and SalariedCommissionEmployee member-function implementations, so we reuse the versions of Figs. 10.2 and 10.5.

Runtime Polymorphic Behavior

SE A Now, if we aim a base-class `SalariedEmployee` pointer at a derived-class `SalariedCommissionEmployee` object and use that pointer to call either `earnings` or `toString`, the derived-class object's function will be invoked polymorphically. Figure 10.10 demonstrates this runtime polymorphic behavior. First, lines 11–22 create a `SalariedEmployee` and a `SalariedCommissionEmployee`, then use static binding to get and output their `toString` results. This will help you confirm the dynamic binding results later in the program. Lines 28–40 show again that a `SalariedEmployee` pointer aimed at a `SalariedEmployee` object can be used to invoke `SalariedEmployee` functionality, and a `SalariedCommissionEmployee` pointer aimed at a `SalariedCommissionEmployee` object can be used to invoke `SalariedCommissionEmployee` functionality. Line 43 aims the base-class pointer `salariedPtr` at the derived-class object `salariedCommission`. Line 51 invokes member function `toString` via the base-class pointer. As you can see in the output, the derived-class `salariedCommission` object's `toString` member function is invoked. Declaring the member function `virtual` and invoking it through a base-class pointer or reference causes the program to determine at execution time which function to invoke based on the type of object to which the handle points, rather than on the type of the handle.

[Click here to view code image](#)

```
1 // fig10_10.cpp
2 // Introducing polymorphism, virtual functions
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this will be part of the standard library
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7 using namespace std;
8
9 int main() {
10     // create base-class object
11     SalariedEmployee salaried{"Sue Jones", 50000, 1000}
```

```
12 // create derived-class object
13 SalariedCommissionEmployee salariedComm.
14     "Bob Lewis", 300.0, 5000.0, .04};
15
16 // output objects using static binding
17 cout << fmt::format("{}\n{}:\n{}\n{}\n",
18                     "INVOKING TOSTRING FUNCTION ON",
19                     "DERIVED-CLASS OBJECTS WITH STATIC",
20                     salaried.toString(), // static
21                     salariedCommission.toString());
22
23
24 cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS";
25             << "DERIVED-CLASS OBJECTS WITH DYNAMIC";
26
27 // natural: aim base-class pointer at base-class
28 SalariedEmployee* salariedPtr{&salaried};
29 cout << fmt::format("{}\n{}:\n{}\n",
30                     "CALLING VIRTUAL FUNCTION TO BASE-CLASS");
31             "TO BASE-CLASS OBJECT INVOKES",
32             salariedPtr->toString()); // dynamic
33
34 // natural: aim derived-class pointer at derived-class
35 SalariedCommissionEmployee* salariedComm.
36 cout << fmt::format("{}\n{}{}:\n{}\n",
37                     "CALLING VIRTUAL FUNCTION TO DERIVED-CLASS");
38             "POINTER TO DERIVED-CLASS OBJECT HAS",
39             "FUNCTIONALITY",
40             salariedCommissionPtr->toString());
41
42 // aim base-class pointer at derived-class
43 salariedPtr = &salariedCommission;
44
45 // runtime polymorphism: invokes Salaried
46 // via base-class pointer to derived-class
47 cout << fmt::format("{}\n{}{}:\n{}\n",
48                     "CALLING VIRTUAL FUNCTION TO BASE-CLASS");
```

```
49         "TO DERIVED-CLASS OBJECT INVO:  
50         "FUNCTIONALITY",  
51         salariedPtr->toString()); //  
52     }
```

< >

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH STATIC BINDING:

name: Sue Jones
salary: \$500.00

name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLAS
TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTION
name: Sue Jones
salary: \$500.00

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-C
POINTER TO DERIVED-CLASS OBJECT INVOKES DERIVED-
name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLAS
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FU
name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04



Fig. 10.10 | Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (Part 2 of 2.)

When `salariedPtr` points to a `SalariedEmployee` object, class `SalariedEmployee`'s `toString` function is invoked (line 32). When `salariedPtr` points to a `SalariedCommissionEmployee` object, class `SalariedCommissionEmployee`'s `toString` function is invoked (line 51). So, **the same `toString` call via a base-class pointer to various objects takes on many forms (in this case, two forms)**. This is **runtime polymorphic behavior**.

Do Not Call Virtual Functions from Constructors and Destructors

CG Calling a virtual function from a base class's constructor or destructor **invokes the base-class version**, even if the base-class constructor or destructor is called while creating or destroying a derived-class object. This is not the behavior you expect for virtual functions, so the C++ Core Guidelines recommend that you do not call them from constructors or destructors.¹¹

¹¹. “C.82: Don’t call virtual functions in constructors and destructors.” Accessed February 8, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual>.

11 C++11 `override` Keyword

Err Declaring `SalariedCommissionEmployee`'s `earnings` and `toString` functions using the `override` keyword tells the compiler to check whether the base class has a virtual member function with the **same signature**. If not, the compiler generates an error. This ensures that you override the appropriate base-class function. **It also prevents you from accidentally hiding a base-class function with the same name but a different signature.** So, to help you prevent errors, apply `override` to the prototype of every derived-class function that overrides a virtual base-class function.

 The C++ Core Guidelines state that:

- `virtual` specifically introduces a new virtual function in a hierarchy, and
- `override` specifically indicates that a derived-class function overrides a base-class virtual function.

So, you should use `only virtual or override` in each virtual function's prototype.¹²

¹². “C.128: Virtual functions should specify exactly one of `virtual`, `override`, or `final`.” Accessed January 21, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override>.

10.7.5 `virtual` Destructors

 SE  The C++ Core Guidelines recommend including a **virtual destructor** in *every* class that contains virtual functions.¹³ This helps prevent subtle errors when a derived class has a custom destructor. **In a class that does not have a destructor, the compiler generates one for you, but the generated one is not virtual.** So, in Modern C++, the virtual destructor definition for most classes is written as:

¹³. “C.35: A base class destructor should be either public and `virtual`, or `protected` and `non-virtual`.” Accessed January 25, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-virtual>.

[Click here to view code image](#)

```
virtual ~SalariedEmployee() = default;
```

This enables you to declare the destructor `virtual` and still have the compiler generate a default destructor for the class—via the notation `= default`.

11 10.7.6 `final` Member Functions and Classes

Before C++11, a derived class could override any base-class virtual function. In C++11 and beyond, a virtual function that's declared **`final`**

in its prototype, as in

```
returnType someFunction (parameters) final;
```

SE A cannot be overridden in any derived class. **In a multi-level class hierarchy, this guarantees that the `final` member-function definition will be used by all subsequent direct and indirect derived classes.**

Similarly, before C++11, any existing class could be used as a base class at any point in a hierarchy. In C++11 and beyond, you can declare a class `final` to prevent it from being used as a base class, as in:

```
class MyClass final {
    // class body
};
```

or:

[Click here to view code image](#)

```
class DerivedClass : public BaseClass final {
    // class body
};
```

Err X Attempting to override a `final` member function or inherit from a `final` base class results in a compilation error.

Perf ✅ A benefit of declaring a `virtual` function `final` is that once the compiler knows a `virtual` function cannot be overridden, it can perform various optimizations. For instance, the compiler might be able to determine at compile time the correct function to call.¹⁴ This optimization is called **devirtualization**.¹⁵

14. Godbolt, Matt. “Optimizations in C++ Compilers.” Accessed February 8, 2021. <https://queue.acm.org/detail.cfm?id=3372264>.

15. Although the C++ standard document discusses a number of possible optimizations, it does not require them. They happen at the discretion of the compiler implementers.

There are cases in which the compiler can devirtualize `virtual` function calls even if the `virtual` functions are not declared `final`. For example, sometimes the compiler can recognize at compile time the type of object that will be used at runtime. In this case, a call to a non-`final` `virtual`

function can be bound at compile time.^{16,17}

16. Godbolt, Matt. “Optimizations in C++ Compiler.” Accessed February 8, 2021. <https://queue.acm.org/detail.cfm?id=3372264>.

17. Brand, Sy. “The Performance Benefits of Final Classes.” Accessed February 8, 2021. <https://devblogs.microsoft.com/cppblog/the-performance-benefits-of-final-classes/>.

10.8 Abstract Classes and Pure virtual Functions

SE  There are cases in which it's useful to define classes from which you never intend to instantiate any objects. Such an **abstract class** defines a **common public interface** for the classes derived from it in a **class hierarchy**. Because abstract classes are used as base classes in inheritance hierarchies, we refer to them as **abstract base classes**. Such classes cannot be used to create objects because, as we'll see, abstract classes are missing pieces. Derived classes must define the “missing pieces” before derived-class objects can be instantiated. We build programs with abstract classes in [Sections 10.9, 10.12 and 10.14](#).

Classes that can be used to instantiate objects are called **concrete classes**. Such classes define or inherit implementations of *every* member function they or their base classes declare. A good example of this is [Section 10.2](#)'s shape hierarchy, which begins with abstract base-class Shape. We then have an abstract base-class TwoDimensionalShape with derived concrete classes Circle, Square and Triangle. We also have an abstract base class, ThreeDimensionalShape, with derived concrete classes Cube, Sphere and Tetrahedron. **Abstract base classes are too general to define real objects—we need to be more specific before instantiating objects.** For example, if someone tells you to “draw the two-dimensional shape,” what shape would you draw? **Concrete classes provide the specifics that make it possible to instantiate objects.**

10.8.1 Pure virtual Functions

SE  A class is made abstract by declaring one or more **pure virtual functions**, each specified by placing “= 0” in its function prototype, as in

[Click here to view code image](#)

```
virtual void draw() const = 0; // pure virtual fu
```

SE A The “`= 0`” is a **pure specifier**. Pure virtual functions do not provide implementations. **Each derived class that hopes to instantiate objects must override all of its base class’s pure virtual functions with concrete implementations.** Otherwise, the derived class is also abstract. By contrast, a regular virtual function has an implementation, giving the derived class the *option* of overriding the function or simply inheriting the base class’s implementation. An abstract class also can have data members and concrete functions. As you’ll see in [Section 10.12](#), **an abstract class that has all pure virtual functions is sometimes called a pure abstract class or an interface.**

SE A Pure virtual functions are used when the base class does not know how to implement a function, but all concrete derived classes should implement it. Returning to our earlier `SpaceObjects` example, it does not make sense for the base-class `SpaceObject` to have a `draw` function implementation. There’s no way to draw a generic space object without knowing which specific type of space object is being drawn.

SE A Although we cannot instantiate objects of an abstract base class, **we can declare pointers and references of the abstract-base-class type. These can refer to objects of any concrete classes derived from the abstract base class.** Programs typically use such pointers and references to manipulate derived-class objects polymorphically at runtime.

10.8.2 Device Drivers: Polymorphism in Operating Systems

Polymorphism is particularly effective for implementing layered software systems. For example, in operating systems, each type of physical input/output device could operate quite differently from the others. Even so, commands to read or write data from and to devices, respectively, may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system—

place some bytes from memory onto that device.

An object-oriented operating system could use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The public functions offered by the device drivers are pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of device drivers.

SE A This architecture also allows new devices to be added to a system easily. The user can just plug in the device and install its new device driver. The operating system “talks” to this new device through its device driver, which has the same public member functions as all other device drivers—those defined in the device-driver abstract base class.

10.9 Case Study: Payroll System Using Runtime Polymorphism

In this example, we use an abstract class and runtime polymorphism to perform payroll calculations for two employee types. We create an `Employee` hierarchy to solve the following problem:

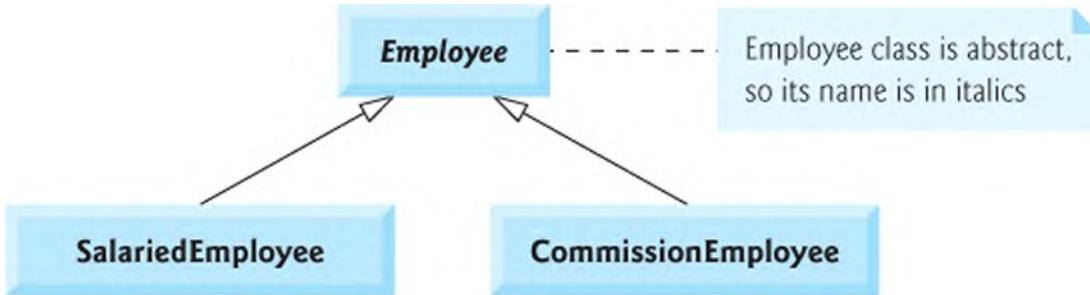
A company pays its employees weekly. The employees are of two types:

1. *Salaried employees are paid a fixed salary regardless of the number of hours worked.*
2. *Commission employees are paid a percentage of their sales.*

The company wants to implement a C++ program that performs its payroll calculations polymorphically.

SE A Many hierarchies start with an abstract base class followed by a row of derived classes that are `final`. We use abstract class `Employee` to represent the *general concept* of an employee and define the interface to the hierarchy—that is, the functions that all `Employee` derived classes must have. The final classes that derive directly from `Employee` are `SalariedEmployee` and `CommissionEmployee`—each of these classes is a **leaf node** in the class hierarchy and cannot be a base class. The

following UML class diagram shows the employee inheritance hierarchy for our runtime polymorphic payroll application. The abstract class name `Employee` is *italicized*, per the UML's convention:



Abstract base-class `Employee` declares the “interface” to the hierarchy —that is, the set of member functions that a program can invoke on *all* `Employee` objects. Each employee, regardless of how earnings are calculated, has a name. So, we chose to define a private data member `m_name` in abstract base class `Employee`.

SE A A derived class can inherit interface and/or implementation from a base class. **Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy.** A base class specifies one or more functions that should be defined by every derived class, but the individual derived classes provide their own implementations of the function(s).

The following subsections implement the `Employee` class hierarchy. The first three each implement one of the abstract or concrete classes. The last implements a test program that builds concrete-class objects and processes them with runtime polymorphism.

10.9.1 Creating Abstract Base-Class `Employee`

Class `Employee` (Figs. 10.11–10.12, discussed in further detail shortly) provides functions `earnings` and `toString`, and *get* and *set* functions that manipulate `Employee`'s `m_name` data member. An `earnings` function certainly applies generally to all `Employees`, but each `earnings` calculation depends on its class. So we declare `earnings` as pure `virtual` in base-class `Employee` because a default implementation does not make sense for that function. There's not enough information to determine what amount `earnings` should return.

Each derived class overrides `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns an employee object's address to a base-class `Employee` pointer, then invokes the object's `earnings` function.

SE A The test program maintains a vector of `Employee` pointers, each of which points to an object that *is an* `Employee`—that is, any object of a concrete derived class of `Employee`. The program iterates through the vector and calls each `Employee`'s `earnings` function. C++ processes these function calls polymorphically. **Including `earnings` as a pure virtual function in `Employee` forces every derived class of `Employee` that wishes to be a concrete class to override `earnings`.**

`Employee`'s `toString` function returns a string containing the `Employee`'s name. As we'll see, each class derived from `Employee` overrides function `toString` to return the `Employee`'s name followed by the rest of the `Employee`'s information. Each derived class's `toString` could also call `earnings`, even though `earnings` is a pure virtual function in base-class `Employee`. Each concrete class is guaranteed to have an `earnings` implementation. Even class `Employee`'s `toString` function can call `earnings`. When you call `toString` through an `Employee` pointer or reference at runtime, you're always calling it on a concrete derived-class object.

The following diagram shows the hierarchy's three classes down the left and functions `earnings` and `toString` across the top. For each class, the diagram shows the desired return value of each function:

	<code>earnings</code>	<code>toString</code>
<code>Employee</code>	<code>pure virtual</code>	<code>name: m_name</code>
<code>Salaried-Employee</code>	<code>m_salary</code>	<code>name: m_name salary: m_salary</code>
<code>Commission-Employee</code>	<code>m_commissionRate * m_grossSales</code>	<code>name: m_name gross sales: m_grossSales commission rate: m_commissionRate</code>

Italic text represents where the values from a particular object are used in the

`earnings` and `toString` functions. Class `Employee` specifies “pure virtual” for function `earnings` to indicate that it’s a pure virtual function with no implementation. Each derived class overrides this function to provide an appropriate implementation. We do not list baseclass `Employee`’s *get* and *set* functions because the derived classes do not override them. Each of these functions is inherited and used “as is” by the derived classes.

[Employee Class Header](#)

Consider class `Employee`’s header (Fig. 10.11). Its public member functions include:

- a constructor that takes the name as an argument (line 9),
- **11** a C++11 defaulted virtual destructor (line 10),
- a *set* function that sets the name (line 12),
- a *get* function that returns the name (line 13),
- pure virtual function `earnings` (line 16), and
- virtual function `toString` (line 17).

Recall that we declared `earnings` as a pure virtual function because, first, we must know the specific `Employee` type to determine the appropriate `earnings` calculation. Each concrete derived class must provide an `earnings` implementation. Then, using a base-class `Employee` pointer or reference, a program can invoke function `earnings` polymorphically for an object of any concrete derived-class of `Employee`.

[Click here to view code image](#)

```
1 // Fig. 10.11: Employee.h
2 // Employee abstract base class.
3 #pragma once // prevent multiple inclusions
4 #include <string>
5 #include <string_view>
6
7 class Employee {
```

```
8 public:
9     explicit Employee(std::string_view name)
10    virtual ~Employee() = default; // compiler-generated
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    // pure virtual function makes Employee an abstract base class
16    virtual double earnings() const = 0; // must be overridden
17    virtual std::string toString() const; // must be overridden
18 private:
19     std::string m_name;
20 };
```



Fig. 10.11 | Employee abstract base class.

Employee Class Member-Function Definitions

Figure 10.12 contains Employee’s member-function definitions. No implementation is provided for virtual function earnings. The virtual function toString implementation (lines 18–20) will be overridden in each derived class. Those derived-class toString functions will call Employee’s toString to get a string containing the information common to all classes in the Employee hierarchy (i.e., the name).

[Click here to view code image](#)

```
1 // Fig. 10.12: Employee.cpp
2 // Abstract-base-class Employee member-function definitions
3 // Note: No definitions are given for pure virtual functions
4 #include "fmt/format.h" // In C++20, this will be replaced by #include <format.h>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // constructor
```

```

9 Employee::Employee(string_view name) : m_name(name)
10
11 // set name
12 void Employee::setName(string_view name) {m_name = name}
13
14 // get name
15 string Employee::getName() const {return m_name}
16
17 // return string representation of an Employee
18 string Employee::toString() const {
19     return fmt::format("name: {}", getName())
20 }

```



Fig. 10.12 | Employee class implementation file.

10.9.2 Creating Concrete Derived-Class SalariedEmployee

Class `SalariedEmployee` (Figs. 10.13–10.14) derives from class `Employee` (Fig. 10.13, line 8). `SalariedEmployee`'s public member functions include:

- a constructor that takes a name and a salary as arguments (line 10),
- **11** a C++11 default virtual destructor (line 11),
- a `set` function to assign a new nonnegative value to data member `m_salary` (line 13) and a `get` function to return `m_salary`'s value (line 14),
- an `override` of `Employee`'s virtual function `earnings` that calculates a `SalariedEmployee`'s earnings (line 17), and
- an `override` of `Employee`'s virtual function `toString` (line 18) that returns a `SalariedEmployee`'s string representation.

[Click here to view code image](#)

```

1 // Fig. 10.13: SalariedEmployee.h
2 // SalariedEmployee class derived from Employee

```

```

3 #pragma once
4 #include <string> // C++ standard string cl
5 #include <string_view>
6 #include "Employee.h" // Employee class def
7
8 class SalariedEmployee final : public Employee
9 {
10     SalariedEmployee(std::string_view name, const std::string& id);
11     virtual ~SalariedEmployee() = default; /
12
13     void setSalary(double salary);
14     double getSalary() const;
15
16     // keyword override signals intent to ov
17     double earnings() const override; // cal
18     std::string toString() const override; /
19 private:
20     double m_salary{0.0};
21 };

```



Fig. 10.13 | SalariedEmployee class header. (Part 1 of 2.)

SalariedEmployee Class Member-Function Definitions

Figure 10.14 defines SalariedEmployee’s member functions:

- Its constructor passes the name argument to the Employee constructor (line 10) to initialize the inherited private data member that’s not directly accessible in the derived class.
- Function earnings (line 28) overrides Employee’s pure virtual function earnings to provide a concrete implementation that returns the weekly salary. If we did not override earnings, SalariedEmployee would inherit Employee’s pure virtual earnings function and would be abstract.
- SalariedEmployee’s toString function (lines 31–34) overrides Employee’s toString. If it did not, the class would inherit

Employee's version that returns a string containing only the employee's name. SalariedEmployee's `toString` returns a string containing the `Employee::toString()` result and the Salaried-Employee's salary.

SalariedEmployee's header declared member functions `earnings` and `toString` as `override` to ensure that we correctly override them. Recall that these are `virtual` in base-class `Employee`, so they remain `virtual` throughout the class hierarchy.

[Click here to view code image](#)

```
1 // Fig. 10.14: SalariedEmployee.cpp
2 // SalariedEmployee class member-function definitions
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be part of the standard library
5 #include "SalariedEmployee.h" // SalariedEmployee.h header file
6 using namespace std;
7
8 // constructor
9 SalariedEmployee::SalariedEmployee(string_view name)
10    : Employee{name} {
11    setSalary(salary);
12 }
13
14 // set salary
15 void SalariedEmployee::setSalary(double salary) {
16    if (salary < 0.0) {
17        throw invalid_argument("Weekly salary must be positive");
18    }
19
20    m_salary = salary;
21 }
22
23 // return salary
24 double SalariedEmployee::getSalary() const
25
```

```
26 // calculate earnings;
27 // override pure virtual function earnings
28 double SalariedEmployee::earnings() const {
29
30     // return a string representation of SalariedEmployee
31     string SalariedEmployee::toString() const {
32         return fmt::format("{}\n{}: ${:.2f}", Employee::toString(),
33                            "salary", getSalary());
34 }
```



Fig. 10.14 | SalariedEmployee class implementation file. (Part 2 of 2.)

10.9.3 Creating Concrete Derived-Class CommissionEmployee

Class `CommissionEmployee` (Figs. 10.15–10.16) derives from `Employee` (Fig. 10.15, line 8). The member-function implementations in Fig. 10.16 include:

- a constructor (lines 9–13) that takes a name, sales amount and commission rate, then passes the name to `Employee`'s constructor (line 10) to initialize the inherited data members,
- *set* functions (lines 16–22 and 28–34) to assign new values to data members `m_grossSales` and `m_commissionRate`,
- *get* functions (lines 25 and 37–39) that return the values of `m_grossSales` and `m_commissionRate`,
- an override of `Employee`'s `earnings` function (lines 42–44) that calculates a `CommissionEmployee`'s earnings, and
- an override of `Employee`'s `toString` function (lines 47–51) that returns a string containing the `Employee::toString()` result, the gross sales and the commission rate.

[Click here to view code image](#)

```
1 // Fig. 10.15: CommissionEmployee.h
2 // CommissionEmployee class derived from Employee
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class CommissionEmployee final : public Employee
9 public:
10     CommissionEmployee(std::string_view name,
11                         double commissionRate);
12     virtual ~CommissionEmployee() = default;
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19
20     // keyword override signals intent to override
21     double earnings() const override; // calculates earnings
22     std::string toString() const override; // returns string representation
23 private:
24     double m_grossSales{0.0};
25     double m_commissionRate{0.0};
26 }
```

Fig. 10.15 | CommissionEmployee class header. (Part 1 of 2.)

[Click here to view code image](#)

```
1 // Fig. 10.16: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be <format>
```

```
5 #include "CommissionEmployee.h" // CommissionEmployee class definition
6 using namespace std;
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(string name,
10                                         double grossSales, double commissionRate)
11 {setGrossSales(grossSales);
12  setCommissionRate(commissionRate);}
13
14
15 // set gross sales amount
16 void CommissionEmployee::setGrossSales(double grossSales)
17 {if (grossSales < 0.0) {
18     throw invalid_argument("Gross sales must be non-negative");
19 }
20
21     m_grossSales = grossSales;
22 }
23
24 // return gross sales amount
25 double CommissionEmployee::getGrossSales() const
26 {return m_grossSales;}
27
28 // set commission rate
29 void CommissionEmployee::setCommissionRate(double commissionRate)
30 {if (commissionRate <= 0.0 || commissionRate > 1.0) {
31     throw invalid_argument("Commission rate must be between 0.0 and 1.0");
32 }
33
34     m_commissionRate = commissionRate;
35 }
36
37 // return commission rate
38 double CommissionEmployee::getCommissionRate() const
39 {return m_commissionRate;}
40
41 // calculate earnings
```

```

42     double CommissionEmployee::earnings() const
43         return getGrossSales() * getCommissionRate();
44     }
45
46     // return string representation of CommissionEmployee
47     string CommissionEmployee::toString() const
48         return fmt::format("{}\n{}: ${:.2f}\n{}":
49                         "gross sales", getGrossSales(),
50                         "commission rate", getCommissionRate());
51     }

```



Fig. 10.16 | CommissionEmployee class implementation file. (Part 1 of 2.)

10.9.4 Demonstrating Runtime Polymorphic Processing

To test our Employee hierarchy, the program in Fig. 10.17 creates an object of each concrete class—SalariedEmployee and CommissionEmployee. The program manipulates these objects first via their object names, then with runtime polymorphism, using a vector of Employee base-class pointers. Lines 17–18 create objects of each concrete derived class. Lines 21–24 output each employee’s information and earnings. **These lines use variable-name handles, so the compiler can identify at compile-time each object’s type to determine which `toString` and `earnings` functions to call.**

[Click here to view code image](#)

```

1  // fig10_17.cpp
2  // Processing Employee derived-class object
3  // then polymorphically using base-class pointer
4  #include <iostream>
5  #include <vector>
6  #include "fmt/format.h" // In C++20, this will be #include <format>
7  #include "Employee.h"
8  #include "SalariedEmployee.h"

```

```
9 #include "CommissionEmployee.h"
10 using namespace std;
11
12 void virtualViaPointer(const Employee* base)
13 void virtualViaReference(const Employee& base)
14
15 int main() {
16     // create derived-class objects
17     SalariedEmployee salaried{"John Smith",
18     CommissionEmployee commission{"Sue Jones",
19
20     // output each Employee
21     cout << "EMPLOYEES PROCESSED INDIVIDUALLY"
22         << fmt::format("{}\n{}{:.2f}\n\n{}", \
23                         salaried.toString(), "earned",
24                         commission.toString(), "earned");
25
26     // create and initialize vector of base-
27     vector<Employee*> employees{&salaried, &
28
29     cout << "EMPLOYEES PROCESSED POLYMORPHIC."
30
31     // call virtualViaPointer to print each :
32     // and earnings using dynamic binding
33     cout << "VIRTUAL FUNCTION CALLS MADE VIA
34
35     for (const Employee* employeePtr : employees)
36         virtualViaPointer(employeePtr);
37     }
38
39     // call virtualViaReference to print each :
40     // and earnings using dynamic binding
41     cout << "VIRTUAL FUNCTION CALLS MADE VIA
42
43     for (const Employee* employeePtr : employees)
44         virtualViaReference(*employeePtr); //
45 }
```

```
46    }
47
48    // call Employee virtual functions toString
49    // base-class pointer using dynamic binding
50    void virtualViaPointer(const Employee* baseClassPtr)
51        cout << fmt::format("{}\nearned ${:.2f}\n",
52                            baseClassPtr->toString(), baseClassPtr->getSalary());
53    }
54
55    // call Employee virtual functions toString
56    // base-class reference using dynamic binding
57    void virtualViaReference(const Employee& baseClassRef)
58        cout << fmt::format("{}\nearned ${:.2f}\n",
59                            baseClassRef.toString(), baseClassRef.getSalary());
60}
```

< >

EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTER

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06

```
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFER
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Fig. 10.17 | Processing Employee derived-class objects with variable-name handles then polymorphically using base-class pointers and references. (Part 3 of 3.)

Creating a vector of Employee Pointers

Line 27 creates and initializes the vector `employees`, which contains two `Employee` pointers aimed at the objects `salaried` and `commission`, respectively. The compiler allows the elements to be initialized with these objects' addresses because a `SalariedEmployee` is an `Employee` and a `CommissionEmployee` is an `Employee`.

Function virtualViaPointer

SE A Lines 35–37 traverse the vector `employees` and invoke function `virtualViaPointer` (lines 50–53) with each element as an argument. Function `virtualViaPointer` receives in its parameter `baseClassPtr` the address stored in a given `employees` element, then uses the pointer to invoke virtual functions `toString` and `earnings`. The function does not contain any `SalariedEmployee` or `CommissionEmployee` type information—it knows only about base-class `Employee`. The program repeatedly aims `baseClassPtr` at different concrete derived-class objects, so the compiler cannot know which concrete class's functions to call through `baseClassPtr`—it must resolve these calls at runtime using dynamic binding. At execution time,

each `virtual`-function call correctly invokes the function on the object to which `baseClassPtr` currently points. The output shows that each class's appropriate functions are invoked and that each object's correct information is displayed. Obtaining each `Employee`'s earnings via runtime polymorphism produces the same results as shown in lines 23 and 24.

Function `virtualViaReference`

Lines 43–45 traverse `employees` and invoke function `virtualViaReference` (lines 57–60) with each element as an argument. Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee&`) a reference to the object obtained by dereferencing the pointer stored in an element of vector `employees` (line 44). Each call to this function invokes virtual functions `toString` and `earnings` via `baseClassRef` to demonstrate that runtime polymorphic processing occurs with base-class references as well. Each `virtual` function invocation calls the function on the object to which `baseClassRef` refers at runtime. This is another example of dynamic binding. The output produced using base-class references is identical to the output produced using base-class pointers and via static binding earlier in the program.

10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

Let's consider how C++ can implement runtime polymorphism, virtual functions and dynamic binding internally. This will give you a solid understanding of how these capabilities can work. More importantly, **you'll appreciate the overhead of runtime polymorphism—in terms of additional memory consumption and processor time**. This can help you determine when to use runtime polymorphism and when to avoid it. **C++ standard-library classes generally are implemented without virtual functions to avoid the associated execution-time overhead and achieve optimal performance.**

 First, we'll explain the data structures that the compiler can build at compile time to support polymorphism at execution time. You'll see that this

can be accomplished through three levels of pointers, i.e., *triple indirection*. Then we'll show how an executing program can use these data structures to execute virtual functions and achieve the dynamic binding associated with polymorphism. Our discussion explains a possible implementation.

Virtual-Function Tables

When C++ compiles a class with one or more virtual functions, it builds a **virtual function table (vtable)** for that class. The *vtable* contains pointers to the class's virtual functions. A **pointer to a function** contains the starting address in memory of the code that performs the function's task. Just as an array name is implicitly convertible to the address of the array's first element, a function name is implicitly convertible to the starting address of its code.

With dynamic binding, an executing program uses a class's *vtable* to select the proper function implementation each time a virtual function is called on an object of that class. The leftmost column of Fig. 10.18 illustrates the *vtables* for the classes Employee, SalariedEmployee and CommissionEmployee.

Employee Class vtable

In the Employee class *vtable*, the first function pointer is set to 0 (i.e., nullptr) because function earnings is a pure virtual function—it does not have an implementation. The second function pointer points to function toString, which returns a string containing the employee's name. We've abbreviated each toString function's output in this figure to conserve space. **Any class with one or more pure virtual functions (represented with the value 0) in its vtable is an abstract class.** Classes SalariedEmployee and CommissionEmployee, which have no nullptrs in their *vtables*, are concrete classes.

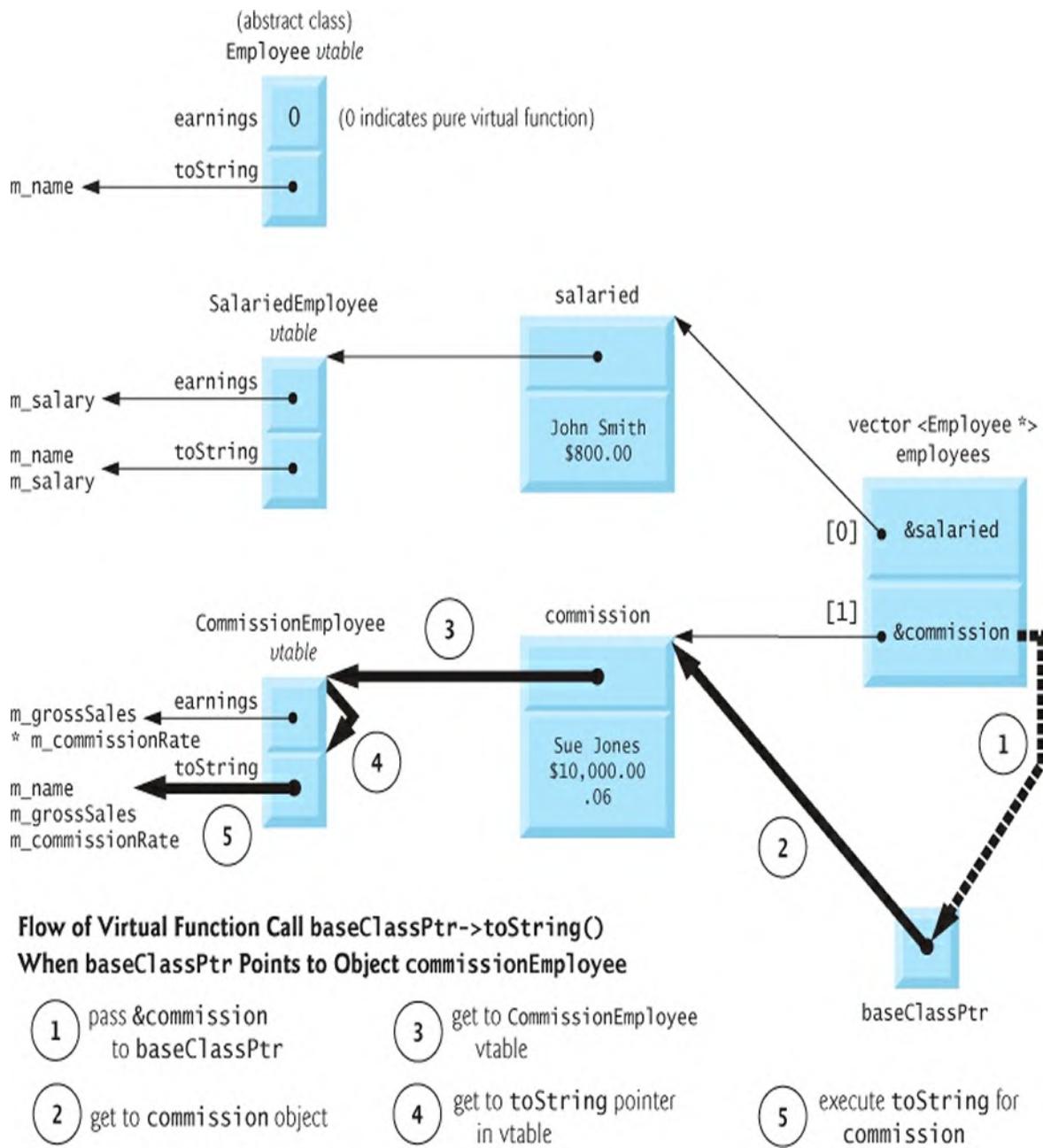


Fig. 10.18 | How virtual function calls work.

SalariedEmployee Class vtable

Class **SalariedEmployee** overrides function **earnings** to return the employee's salary, so the function pointer points to **SalariedEmployee**'s **earnings** function, which returns the salary. **SalariedEmployee** also overrides **toString**, so the corresponding function pointer points to **SalariedEmployee**'s **toString** function, which returns the

employee's name and salary.

CommissionEmployee Class vtable

The `earnings` function pointer in the `CommissionEmployee vtable` points to the `CommissionEmployee`'s `earnings` function, which returns the employee's gross sales multiplied by the commission rate. The `toString` function pointer points to the `CommissionEmployee` version of the function, which returns the employee's name, commission rate and gross sales. As in class `SalariedEmployee`, both functions override class `Employee`'s functions.

Inheriting Concrete virtual Functions

In our `Employee` case study, each concrete class provides its own `virtual earnings` and `toString` implementations. You've learned that, because `earnings` is a pure virtual function, each *direct* derived class of `Employee` must implement `earnings` to be a concrete class. Direct derived classes do not need to implement `toString` to be considered concrete—they can inherit class `Employee`'s `toString` implementation. In our case, both derived classes override `Employee`'s `toString`.

If a derived class in our hierarchy were to inherit `toString` and not override it, this function's `vtable` pointer would simply point to the inherited implementation. For example, if `CommissionEmployee` did not override `toString`, `CommissionEmployee`'s `toString` function pointer in the `vtable` would point to the same `toString` function as in class `Employee`'s `vtable`.

Three Levels of Pointers to Implement Runtime Polymorphism

Runtime polymorphism can be accomplished through an elegant data structure involving three levels of pointers. We've discussed one level—the function pointers in the `vtable`. These point to the actual functions that execute when a virtual function is invoked.

Now we consider the second level of pointers. **Whenever an object with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the `vtable` for that class.** This pointer is normally at the front of the object, but it isn't required to be implemented that way. In the `vtable` diagram, these pointers are associated with the `SalariedEmployee`

and `CommissionEmployee` objects defined in Fig. 10.17. The diagram shows each object's data member values.

The third level of pointers are handles to the objects on which the virtual function will be called. The *vtable* diagram depicts the vector `employees` containing `Employee` pointers.

Now let's see how a typical virtual function call executes. Consider in the function `virtualViaPointer` the call `baseClassPtr->toString()` (Fig. 10.17, line 52). Assume that `baseClassPtr` contains `employees[1]`, commission's address in `employees`. When this statement is compiled, the compiler sees that the call is made via a base-class pointer and that `toString` is a virtual function. The compiler sees that `toString` is the second entry in each *vtable*. So it includes an **offset** into the table of machine-language object-code pointers to find the code that will execute the virtual function call.

The compiler generates code that performs the following operations—the numbers in the list correspond to the circled numbers in Fig. 10.18:

1. Select the *i*th `employees` entry—the `commission` object's address—and pass it as an argument to function `virtualViaPointer`. This aims parameter `base-ClassPtr` at the `commission` object.
2. Dereference that pointer to get to the `commission` object, which begins with a pointer to class `CommissionEmployee`'s *vtable*.
3. Dereference the *vtable* pointer to get to class `CommissionEmployee`'s *vtable*.
4. Skip the offset of eight bytes to select the `toString` function pointer.¹⁸

¹⁸. In practice, Steps 3 and 4 can be implemented as a single machine instruction.

5. Execute `toString` for the `commission` object, returning a string containing the employee's name, gross sales and commission rate.

Perf  The *vtable* diagram's data structures may appear complex, but this complexity is managed by the compiler and hidden from you, making runtime polymorphic programming straightforward. The pointer dereferencing operations and memory accesses for each virtual function

call require additional execution time, and the *vtables* and *vtable* pointers added to the objects require some additional memory.

 **Runtime polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient.** For most applications, you can use these capabilities with nominal impact on execution performance and memory consumption. In some situations, polymorphism's overhead may be too high—such as in real-time applications with stringent execution-timing performance requirements, or in an application with an enormous number of *small* objects in which the size of the *vtable* pointer is large when compared to the size of each object.

10.11 Non-Virtual Interface (NVI) Idiom

The **non-virtual interface idiom (NVI)**^{19,20} is another way to implement runtime polymorphism using class hierarchies. The idiom was first proposed by Herb Sutter in his paper, “Virtuality.”²¹ Sutter lists four guidelines for implementing class hierarchies, each of which we’ll use in this example:

19. “Non-virtual interface (NVI) pattern.” Accessed February 8, 2021. <https://w.wiki/yRF>.
20. Marius Bancila. *Modern C++ Programming Cookbook: Master C++ Core Language and Standard Library Features, with over 100 Recipes, Updated to C++20*, pp. 562–67. Birmingham: Packt Publishing, 2020.
21. Sutter, Herb. “Virtuality,” C/C++ Users Journal, vol. 19, no. 9, September 2001. Accessed February 16, 2021. <http://www.gotw.ca/publications/mill18.htm>.
1. “Prefer to make interfaces non-virtual, using Template Method”—an object-oriented design pattern.^{22,23} Sutter explains that a public virtual function serves *two* purposes—it describes part of a class’s interface, and it enables derived classes to customize behavior by overriding the virtual function. He recommends that each function should serve only *one* purpose.
22. “Template Method Pattern.” Accessed February 3, 2021. <https://w.wiki/yFd>.
23. Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, pp. 325–330.
2. “Prefer to make virtual functions private.” **A derived class can override its base class’s `private` virtual functions.** By making

a virtual function private, it serves one purpose—enabling derived classes to customize behavior by overriding the virtual function. In the base class, a non-virtual function invokes the private virtual function internally as an implementation detail.

3. “Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.” This enables derived classes to override the base-class virtual function and take advantage of its base-class implementation to avoid duplicating code in the derived class.
4. “A base-class destructor should be either public and virtual, or protected and non-virtual.” Here, we’ll again make the Employee base class’s destructor public and virtual. In [Chapter 11](#), you’ll see that public virtual destructors are important when deleting dynamically allocated derived-class objects via base-class pointers.

 Sutter’s paper demonstrates each recommendation, focusing on separating a class’s interface from its implementation as a good software-engineering practice.²⁴

²⁴. For some related inheritance discussions, see the ISO C++ FAQ, “Inheritance — What your mother never told you” at <https://isocpp.org/wiki/faq/strange-inheritance>.

Refactoring Class Employee for the NVI Idiom

Per Sutter’s guidelines, let’s refactor our Employee class (Figs. 10.19–10.20). Most of the code is identical to [Section 10.9](#), so we focus here on only the changes. There are several key changes in [Fig. 10.19](#):

- Employee’s public earnings (line 15) and toString (line 16) member functions are no longer declared virtual, and, as you’ll see, earnings will have an implementation. Functions earnings and toString each now serve *one* purpose—to allow client code to get an Employee’s earnings and string representation, respectively. Their original second purpose as customization points for derived classes will now be implemented via new member functions.
- We added the protected virtual function getString (line 18) to serve as a customization point for derived classes. This function is

called by the non-virtual `toString` function. As you'll see, our derived classes' `getString` functions will both override and call class `Employee`'s `getString`.

- We added the `private pure virtual` function `getPay` (line 21) to serve as a customization point for derived classes. This function is called by the non-virtual `earnings` function. Derived classes override `getPay` to specify custom earnings calculations.

[Click here to view code image](#)

```
1 // Fig. 10.19: Employee.h
2 // Employee abstract base class.
3 #pragma once // prevent multiple inclusions
4 #include <string>
5 #include <string_view>
6
7 class Employee {
8 public:
9     Employee(std::string_view name);
10    virtual ~Employee() = default;
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    double earnings() const; // not virtual
16    std::string toString() const; // not vir
17 protected:
18    virtual std::string getString() const; /
19 private:
20    std::string m_name;
21    virtual double getPay() const = 0; // pu
22};
```



Fig. 10.19 | Employee abstract base class.

There are several key changes in class `Employee`'s member-function implementations (Fig. 10.20):

- Member-function `earnings` (line 18) now provides a concrete implementation that returns the result of calling the private pure virtual function `getPay`.
- Member-function `toString` (line 21) now returns the result of calling the protected virtual function `getString`.
- We now define the protected member-function `getString` to specify an `Employee`'s default string representation containing an `Employee`'s name. This function is protected so derived classes can both override it *and* call it to get the base-class part of the derived-class string representations.

[Click here to view code image](#)

```
1 // Fig. 10.20: Employee.cpp
2 // Abstract-base-class Employee member-func
3 // Note: No definitions are given for pure
4 #include "fmt/format.h" // In C++20, this w
5 #include "Employee.h" // Employee class def
6 using namespace std;
7
8 // constructor
9 Employee::Employee(string_view name) : m_na
10
11 // set name
12 void Employee::setName(string_view name) {m_
13
14 // get name
15 string Employee::getName() const {return m_
16
17 // public non-virtual function; returns Emp
18 double Employee::earnings() const {return g
19
20 // public non-virtual function; returns Emp
```

```
21     string Employee::toString() const {return g...
22
23     // protected virtual function that derived ...
24     string Employee::getString() const {
25         return fmt::format("name: {}", getName())
26     }
```



Fig. 10.20 | Abstract-base-class `Employee` member-function definitions.

Updated Class `SalariedEmployee`

Our refactored `SalariedEmployee` class (Figs. 10.21–10.22) has several key changes:

- The class's header (Fig. 10.21) no longer contains prototypes for the `earnings` and `toString` member functions. These public non-virtual base-class functions are now inherited by class `SalariedEmployee`.
- The class's private section now declares overrides for the base-class's private pure virtual function `getPay` and protected virtual function `getString` (Fig. 10.21, lines 19–20). We made `getString` private here because this class is `final`, so no other classes can derive from it.
- The class's member-function implementations (Fig. 10.22) now include the overridden private `getPay` function (line 28), which returns the salary, and private `getString` function (lines 31–34), which specifies a `SalariedEmployee`'s string representation. Note that `SalariedEmployee`'s `getString` calls class `Employee`'s protected `getString` to get part of the string representation.

[Click here to view code image](#)

```
1  // Fig. 10.21: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee
3  #pragma once
```

```
4 #include <string> // C++ standard string class
5 #include <string_view>
6 #include "Employee.h" // Employee class definition header
7
8 class SalariedEmployee final : public Employee {
9 public:
10     SalariedEmployee(std::string_view name, double salary);
11     virtual ~SalariedEmployee() = default; // destructor
12
13     void setSalary(double salary);
14     double getSalary() const;
15 private:
16     double m_salary{0.0};
17
18     // keyword override signals intent to override
19     double getPay() const override; // calculate pay
20     std::string getString() const override;
21 };
```



Fig. 10.21 | SalariedEmployee class derived from Employee.

[Click here to view code image](#)

```
1 // Fig. 10.22: SalariedEmployee.cpp
2 // SalariedEmployee class member-function definitions
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be <format>
5 #include "SalariedEmployee.h" // SalariedEmployee header file
6 using namespace std;
7
8 // constructor
9 SalariedEmployee::SalariedEmployee(string_view name,
10     : Employee{name} {
11     setSalary(salary);
12 }
```

```

13
14 // set salary
15 void SalariedEmployee::setSalary(double sal-
16     if (salary < 0.0) {
17         throw invalid_argument("Weekly salary
18     }
19
20     m_salary = salary;
21 }
22
23 // return salary
24 double SalariedEmployee::getSalary() const
25
26 // calculate earnings;
27 // override pure virtual function getPay in
28 double SalariedEmployee::getPay() const { re-
29
30 // return a string representation of Salari-
31 string SalariedEmployee::getString() const
32     return fmt::format("{}\n{}: ${:.2f}", Em-
33             "salary", getSalary());
34 }

```

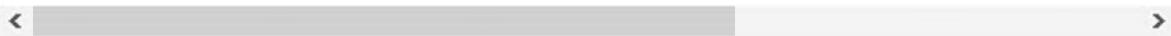


Fig. 10.22 | SalariedEmployee class member-function definitions.
(Part 1 of 2.)

Updated Class CommissionEmployee

Our refactored CommissionEmployee class (Figs. 10.23–10.24) has several key changes:

- The class's header (Fig. 10.23) no longer contains prototypes for the earnings and `toString` member functions. These public non-virtual base-class functions are now inherited by class `CommissionEmployee`.
- The class's private section now declares overrides for the base-class's private pure virtual function `getPay` and protected

virtual function `getString` (Fig. 10.23, lines 24–25). Again, we made `getString` private because this class is final, so no other classes can derive from it.

- The class’s member-function implementations (Fig. 10.24) now include the overridden `getPay` function (lines 42–44), which returns the result of the commission calculation, and `getString` function (lines 47–51), which specifies a `CommissionEmployee`’s string representation. `CommissionEmployee`’s `get-String` calls class `Employee`’s protected `getString` to get part of the string representation.

[Click here to view code image](#)

```
1 // Fig. 10.23: CommissionEmployee.h
2 // CommissionEmployee class derived from Employee
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class CommissionEmployee final : public Employee
9 {
10     CommissionEmployee(std::string_view name,
11                         double commissionRate);
12     virtual ~CommissionEmployee() = default;
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19 private:
20     double m_grossSales{0.0};
21     double m_commissionRate{0.0};
22
23     // keyword override signals intent to override
24     double getPay() const override; // calculate pay
```

```
25     std::string getString() const override;
26 }
```



Fig. 10.23 | CommissionEmployee class derived from Employee.
(Part 1 of 2.)

[Click here to view code image](#)

```
1 // Fig. 10.24: CommissionEmployee.cpp
2 // CommissionEmployee class member-function
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this w
5 #include "CommissionEmployee.h" // Commissi
6 using namespace std;
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(strin
10     double grossSales, double commissionRate
11     setGrossSales(grossSales);
12     setCommissionRate(commissionRate);
13 }
14
15 // set gross sales amount
16 void CommissionEmployee::setGrossSales(double
17     if (grossSales < 0.0) {
18         throw invalid_argument("Gross sales m
19     }
20
21     m_grossSales = grossSales;
22 }
23
24 // return gross sales amount
25 double CommissionEmployee::getGrossSales() {
26
27 // set commission rate
```

```

28 void CommissionEmployee::setCommissionRate(
29     if (commissionRate <= 0.0 || commissionRate > 1.0)
30         throw invalid_argument("Commission rate must be between 0.0 and 1.0");
31     }
32
33     m_commissionRate = commissionRate;
34 }
35
36 // return commission rate
37 double CommissionEmployee::getCommissionRate() const {
38     return m_commissionRate;
39 }
40
41 // calculate earnings
42 double CommissionEmployee::getPay() const {
43     return getGrossSales() * getCommissionRate();
44 }
45
46 // return string representation of CommissionEmployee
47 string CommissionEmployee::getString() const {
48     return fmt::format("{}\n{}: ${:.2f}\n{}:\n"
49                         "gross sales", getGrossSales()
50                         "commission rate", getCommissionRate());
51 }
```



Fig. 10.24 | CommissionEmployee class member-function definitions. (Part 1 of 2.)

Runtime Polymorphism with the Employee Hierarchy Using NVI

SE A The test application for this example is identical to the one in Fig. 10.17, so we show only the output in Fig. 10.25. The program's output also is identical to Fig. 10.17, demonstrating that we still get polymorphic processing even with protected and private base-class virtual functions. Our client code now calls only non-virtual functions, yet each derived class was able to provide custom behavior by overriding the

protected and **private** base-class **virtual** functions. The **virtual** functions are now internal implementation details of the class hierarchy, hidden from the client-code programmer. We can change those **virtual** function implementations—and potentially even their signatures—without affecting the client code.

[Click here to view code image](#)

```
EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING
```

```
name: John Smith  
salary: $800.00  
earned $800.00
```

```
name: Sue Jones  
gross sales: $10000.00  
commission rate: 0.06  
earned $600.00
```

```
EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING
```

```
VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
```

```
name: John Smith  
salary: $800.00  
earned $800.00
```

```
name: Sue Jones  
gross sales: $10000.00  
commission rate: 0.06  
earned $600.00
```

```
VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
```

```
name: John Smith  
salary: $800.00  
earned $800.00
```

```
name: Sue Jones  
gross sales: $10000.00
```

```
commission rate: 0.06  
earned $600.00
```

< >

Fig. 10.25 | Processing Employee derived-class objects with static binding, then polymorphically using dynamic binding. (Part 2 of 2.)

10.12 Program to an Interface, Not an Implementation²⁵

²⁵. Defined in Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, pp. 17–18; also discussed in Joshua Bloch. *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

Implementation inheritance is primarily used to define closely related classes with many of the same data members and member function implementations. This kind of inheritance tends to create **tightly coupled** classes in which the base class's data members and member functions are inherited into derived classes. Changes to a base class directly affect all corresponding derived classes.

SE A Tight coupling can make modifying class hierarchies difficult.

Consider how you might modify Section 10.9's Employee hierarchy so that it also supports retirement plans. There are many different retirement plan types, including 401Ks and IRAs. We might add a pure virtual makeRetirementDeposit member function to the class Employee. Then we'd define various derived classes such as SalariedEmployeeWith401K, SalariedEmployeeWithIRA, CommissionEmployeeWith401K, CommissionEmployeeWithIRA, etc., each with an appropriate makeRetirementDeposit implementation. As you can see, you quickly wind up with a proliferation of derived classes, making the hierarchy more challenging to implement and maintain.

Small inheritance hierarchies under the control of one person tend to be more manageable than large ones maintained by many people. This is true even with the tight coupling associated with implementation inheritance.

Rethinking the Employee Hierarchy Using Composition and

Dependency Injection

Over the years, programmers have written numerous papers, articles and blog posts about the problems with tightly coupled class hierarchies like [Section 10.9](#)'s Employee hierarchy. In this example, we'll refactor our Employee hierarchy so that the Employee's compensation model is not "hardwired" into the class hierarchy.²⁶

²⁶. We'd like to thank Brian Goetz, Oracle's Java Language Architect, for suggesting the class architecture we use in this section when he reviewed a recent edition of our book *Java How to Program*.

 **To do so, we'll use composition and dependency injection in which a class contains a pointer to an object that provides a behavior required by objects of the class.** In our Employee payroll example, that behavior is calculating each Employee's earnings. We'll define a new Employee class that *has a* pointer to a CompensationModel object with earnings and toString member functions. This class will not be a base class—to emphasize that we'll make it final. We'll then define derived classes of class CompensationModel that implement how Employees get compensated:

- fixed salary, and
- commission based on gross sales.

Of course, we could define other CompensationModels, too.

Interface Inheritance Is Best for Flexibility

 For our CompensationModels, we'll use **interface inheritance**. Each CompensationModel concrete class will inherit from a class containing **only pure virtual functions**. Such a class is called an **interface** or a **pure abstract class**. The C++ Core Guidelines recommend inheriting from pure abstract classes rather than classes with implementation details.^{27,28,29,30}

²⁷. "I.25: Prefer abstract classes as interfaces to class hierarchies." Accessed February 8, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abstract>.

²⁸. "C.121: If a base class is used as an interface, make it a pure abstract class." Accessed February 8, 2021.

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-abstract>.

29. “C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.” Accessed February 8, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-separation>.
30. “C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance.” Accessed February 8, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-kind>.

 **Interfaces typically do not have data members.** Interface inheritance may require more work than implementation inheritance because concrete classes must provide both data and implementations of the interface’s pure virtual member functions, even if the data and member-function implementations are similar or identical among classes. As we’ll discuss at the end of the example, this approach gives you additional flexibility by eliminating the tight coupling between classes. The discussion of device drivers in the context of abstract classes at the end of [Section 10.8](#) is a good example of how interfaces enable systems to be modified easily.

10.12.1 Rethinking the Employee Hierarchy —CompensationModel Interface

Let’s reconsider [Section 10.9](#)’s Employee hierarchy with composition and an interface. We can say that each Employee *has a* CompensationModel. [Figure 10.26](#) defines the **interface CompensationModel**, which is a pure abstract class, so it does not have a .cpp file. The class has a compiler-generated virtual destructor and two pure virtual functions:

- `earnings` to calculate an employee’s pay, based on its CompensationModel, and
- `toString` to create a string representation of a CompensationModel.

Any class that inherits from CompensationModel and overrides its pure virtual functions *is a* CompensationModel that implements this interface.

[Click here to view code image](#)

```
1 // Fig. 10.26: CompensationModel.h
2 // CompensationModel "interface" is a pure .
3 #pragma once // prevent multiple inclusions
4 #include <string>
5
6 class CompensationModel {
7 public:
8     virtual ~CompensationModel() = default;
9     virtual double earnings() const = 0; // ]
10    virtual std::string toString() const = 0
11};
```



Fig. 10.26 | CompensationModel “interface” is a pure abstract base class.

10.12.2 Class Employee

Figure 10.27 defines our new Employee class. **Each Employee has a pointer to the implementation of its CompensationModel** (line 16). Note that this class has been declared final so that it cannot be used as a base class.

[Click here to view code image](#)

```
1 // Fig. 10.27: Employee.h
2 // An Employee "has a" CompensationModel.
3 #pragma once // prevent multiple inclusions
4 #include <string>
5 #include <string_view>
6 #include "CompensationModel.h"
7
8 class Employee final {
9 public:
10    Employee(std::string_view name, Compensa
```

```
11     void setCompensationModel(CompensationMo
12     double earnings() const;
13     std::string toString() const;
14 private:
15     std::string m_name{};
16     CompensationModel* m_modelPtr{}; // poin
17 };
```



Fig. 10.27 | An Employee “has a” CompensationModel.

Figure 10.28 defines class Employee’s member functions:

- The constructor (lines 11–12) initializes the Employee’s name and aims its CompensationModel pointer at an object that implements the CompensationModel interface. This technique is known as **constructor injection**. The constructor receives a pointer (or reference) to another object and stores it in the object being constructed.
- The setCompensationModel member function (lines 16–18) enables the client code to change an Employee’s CompensationModel by aiming m_modelPtr at a different CompensationModel implementation. This technique is known as **property injection**.
- The earnings member function (lines 21–23) determines the Employee’s earnings by calling the CompensationModel implementation’s earnings member function via the CompensationModel pointer m_modelPtr.
- The toString member function (lines 26–28) creates an Employee’s string representation consisting of the Employee’s name, followed by the compensation information, which we get by calling the CompensationModel implementation’s toString member function via a the CompensationModel pointer m_modelPtr

SE  **Constructor injection and property injection are both forms of dependency injection. You specify part of an object’s behavior by providing it with a pointer or reference to another object that defines the**

behavior.³¹ In this example, a `CompensationModel` provides the behavior that enables an `Employee` to calculate its earnings and to generate a string representation.

[31. "Dependency injection."](#) Accessed February 8, 2021. <https://w.wiki/yQx>.

[Click here to view code image](#)

```
1 // Fig. 10.28: Employee.cpp
2 // Class Employee member-function definitions
3 #include <string>
4 #include "fmt/format.h" // In C++20, this will be #include <format.h>
5 #include "CompensationModel.h"
6 #include "Employee.h"
7 using namespace std;
8
9 // constructor performs "constructor injection"
10 // the CompensationModel pointer to a CompensationModel object
11 Employee::Employee(string_view name, CompensationModel* modelPtr)
12     : m_name{name}, m_modelPtr{modelPtr} {}
13
14 // set function performs "property injection"
15 // CompensationModel pointer to a new CompensationModel object
16 void Employee::setCompensationModel(CompensationModel* modelPtr)
17     m_modelPtr = modelPtr;
18 }
19
20 // use the CompensationModel to calculate total earnings
21 double Employee::earnings() const {
22     return m_modelPtr->earnings();
23 }
24
25 // return string representation of Employee
26 string Employee::toString() const {
27     return fmt::format("{}\n{}", m_name, m_modelPtr);
28 }
```



Fig. 10.28 | Class Employee member-function definitions.

10.12.3 CompensationModel Implementations

Next, let's define our CompensationModel implementations. Objects of these classes will be injected into Employee objects to specify how to calculate their earnings.

Salaried Derived Class of CompensationModel

A Salaried compensation model (Figs. 10.29–10.30) defines how to pay an Employee who receives a fixed salary. The class contains an `m_salary` data member and overrides interface CompensationModel's `earnings` and `toString` member functions. **Salaried is declared final (line 7), so it is a leaf node in the CompensationModel hierarchy. It may not be used as a base class.**

[Click here to view code image](#)

```
1 // Fig. 10.29: Salaried.h
2 // Salaried implements the CompensationMode
3 #pragma once
4 #include <string>
5 #include "CompensationModel.h" // Compensat
6
7 class Salaried final : public CompensationM
8 public:
9     explicit Salaried(double salary);
10    double earnings() const override;
11    std::string toString() const override;
12 private:
13     double m_salary{0.0};
14 }
```



Fig. 10.29 | Salaried implements the CompensationModel interface.

[Click here to view code image](#)

```
1 // Fig. 10.30: Salaried.cpp
2 // Salaried compensation model member-funct
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this w
5 #include "Salaried.h" // class definition
6 using namespace std;
7
8 // constructor
9 Salaried::Salaried(double salary) : m_salary(salary)
10    if (m_salary < 0.0) {
11        throw invalid_argument("Weekly salary must be >= 0");
12    }
13 }
14
15 // override CompensationModel pure virtual
16 double Salaried::earnings() const {return m_salary;}
17
18 // override CompensationModel pure virtual
19 string Salaried::toString() const {
20     return fmt::format("salary: ${:.2f}", m_salary);
21 }
```



Fig. 10.30 | Salaried compensation model member-function definitions.

Commission Derived Class of CompensationModel

A Commission compensation model (Figs. 10.31–10.32) defines how to pay an Employee commission based on gross sales. The class contains `m_grossSales` and `m_commissionRate` data members and overrides interface `CompensationModel`'s `earnings` and `toString` member functions. Like class `Salaried`, class `Commission` is declared `final` (line 7), so it is a leaf node in the `CompensationModel` hierarchy. It may not be used as a base class.

[Click here to view code image](#)

```
1 // Fig. 10.31: Commission.h
2 // Commission implements the CompensationModel
3 #pragma once
4 #include <string>
5 #include "CompensationModel.h" // CompensationModel
6
7 class Commission final : public CompensationModel {
8 public:
9     Commission(double grossSales, double commissionRate)
10    double earnings() const override;
11    std::string toString() const override;
12 private:
13     double m_grossSales{0.0};
14     double m_commissionRate{0.0};
15 };
```



Fig. 10.31 | Commission implements the CompensationModel interface.

[Click here to view code image](#)

```
1 // Fig. 10.32: Commission.cpp
2 // Commission member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be replaced by <format>
5 #include "Commission.h" // class definition
6 using namespace std;
7
8 // constructor
9 Commission::Commission(double grossSales, double commissionRate)
10   : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
11
12     if (m_grossSales < 0.0) {
```

```
13         throw invalid_argument("Gross sales must be positive");
14     }
15
16     if (m_commissionRate <= 0.0 || m_commissionRate > 1.0)
17         throw invalid_argument("Commission rate must be between 0.0 and 1.0");
18     }
19 }
20
21 // override CompensationModel pure virtual
22 double Commission::earnings() const {
23     return m_grossSales * m_commissionRate;
24 }
25
26 // override CompensationModel pure virtual
27 string Commission::toString() const {
28     return fmt::format("gross sales: ${:.2f}\n"
29                         "m_grossSales, m_commissionRate");
30 }
```



Fig. 10.32 | Commission member-function definitions.

10.12.4 Testing the New Hierarchy

We've created our `CompensationModel` **interface** and derived-class implementations defining how Employees get paid. Now, let's create `Employee` objects and initialize each with an appropriate concrete `CompensationModel` implementation (Fig. 10.33).

[Click here to view code image](#)

```
1 // fig10_33.cpp
2 // Processing Employees with various Compensation Models
3 #include <iostream>
4 #include <vector>
5 #include "fmt/format.h" // In C++20, this will be part of the standard library
6 #include "Employee.h"
```

```
7 #include "Salaried.h"
8 #include "Commission.h"
9 using namespace std;
10
11 int main() {
12     // create CompensationModels and Employee
13     Salaried salaried{800.0};
14     Employee salariedEmployee{"John Smith",
15
16     Commission commission{10000, .06};
17     Employee commissionEmployee{"Sue Jones",
18
19     // create and initialize vector of Employee
20     vector employees{salariedEmployee, commis-
21
22     // print each Employee's information and
23     for (const Employee& employee : employee)
24         cout << fmt::format("{}\nearned: ${}..",
25                             employee.toString(), emplo-
26
27 }
```



```
John Smith
salary: $800.00
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 10.33 | Processing Employees with various CompensationModels.

Line 13 creates a `Salaried` compensation model object (`salaried`), then line 14 creates the `Employee` `salariedEmployee` and injects its

`CompensationModel`, passing as the second constructor argument a pointer to `salaried`. Line 16 creates a `Commission` compensation model object (`commission`), then line 17 creates the `Employee` `commissionEmployee` and injects its `CompensationModel`, passing as the second constructor argument a pointer to `commission`. Line 20 creates a vector of `Employees` and initializes it with the `salariedEmployee` and `commissionEmployee` objects. Finally, lines 23–26 iterate through the vector, displaying each `Employee`'s string representation and earnings.

Flexibility If CompensationModels Change

Declaring the `CompensationModels` as separate classes that implement the same interface provides flexibility for future changes. Suppose a company adds new ways to pay employees. We can simply define a new `CompensationModel` derived class with an appropriate earnings function.

Flexibility If Employees Are Promoted

 **SE** The interface-based composition and dependency-injection approach we used in this example is more flexible than [Section 10.9](#)'s class hierarchy. In [Section 10.9](#), if an `Employee` were promoted, you'd need to change its object type by creating a new object of the appropriate `Employee` derived class, then moving data into the new object. **Using dependency injection, you can simply call `Employee`'s `setCompensationModel` member function and inject a pointer to a different `CompensationModel` that replaces the existing one.**

Flexibility If Employees Acquire New Capabilities

The interface-based composition and dependency-injection approach is also more flexible for enhancing class `Employee`. If we decide to support retirement plans (such as 401Ks and IRAs), we could say that every `Employee` *has a* `RetirementPlan`. First, we'd define interface `RetirementPlan` with a `makeRetirementDeposit` member function and provide appropriate derived-class implementations.

Using interface-based composition and dependency-injection, as shown in this example, requires only small changes to class `Employee` to support

RetirementPlans:

- a data member that points to a RetirementPlan,
- one more constructor argument to initialize the RetirementPlan pointer, and
- a setRetirementPlan member function we can call if we ever need to change the RetirementPlan.

10.13 Runtime Polymorphism with `std::variant` and `std::visit`

17 SE  So far, we've achieved runtime polymorphism via implementation inheritance or interface inheritance. As you've seen, both techniques require class hierarchies. What if you have objects of *unrelated* classes, but you'd still like to process those objects polymorphically at runtime. You can achieve this with C++17's **class template `std::variant`** and the **standardlibrary function `std::visit`** (both in header `<variant>`).^{32,33,34,35} **The caveat is that you must know in advance all the types your program needs to process via runtime polymorphism—known as a closed set of types.** A `std::variant` object can store one object at a time of any type specified when you create the `std::variant` object. As you'll see, you call functions on the objects in a `std::variant` object using the `std::visit` function.

32. Nevin Liber, “The Many Variants of `std::variant`,” YouTube Video, June 16, 2019, <https://www.youtube.com/watch?v=JUxhwf7gYLg>.

33. “`std::variant`.” Accessed January 30, 2021. <https://en.cppreference.com/w/cpp/utility/variant>.

34. Bartłomiej Filipek, “Runtime Polymorphism with `std::variant` and `std::visit`.” Accessed January 30, 2021. <https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html>.

35. Bartłomiej Filipek, “Everything You Need to Know About `std::variant` from C++17.” Accessed January 30, 2021. <https://www.bfilipek.com/2018/06/variant.html>.

To demonstrate runtime polymorphism with `std::variant`, we'll reimplement our [Section 10.12](#) example. The classes used here are nearly identical, so we'll point out only the differences.

Compensation Model Salaried

Class `Salaried` (Figs. 10.34–10.35) defines the compensation model for an Employee who gets paid a fixed salary. The only difference between this class and the one in [Section 10.12](#) is that **this `Salaried` is not a derived class. So, its `earnings` and `toString` member functions do not override base-class `virtual` functions.**

[Click here to view code image](#)

```
1 // Fig. 10.34: Salaried.h
2 // Salaried compensation model.
3 #pragma once
4 #include <string>
5
6 class Salaried {
7 public:
8     Salaried(double salary);
9     double earnings() const;
10    std::string toString() const;
11 private:
12     double m_salary{0.0};
13 }
```

Fig. 10.34 | Salaried compensation model.

[Click here to view code image](#)

```
1 // Fig. 10.35: Salaried.cpp
2 // Salaried compensation model member-funct.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this w.
5 #include "Salaried.h" // class definition
6 using namespace std;
7
8 // constructor
```

```

9  Salaried::Salaried(double salary) : m_salary(salary)
10 if (m_salary < 0.0) {
11     throw invalid_argument("Weekly salary must be non-negative");
12 }
13 }
14
15 // calculate earnings
16 double Salaried::earnings() const {return m_salary * 40.0;}
17
18 // return string containing Salaried compensation information
19 string Salaried::toString() const {
20     return fmt::format("salary: ${:.2f}", m_salary);
21 }
```



Fig. 10.35 | Salaried compensation model member-function definitions.
(Part 1 of 2.)

Compensation Model Commission

Class `Commission` (Figs. 10.36–10.37) defines the compensation model for an `Employee` who gets paid commission based on gross sales. Like `Salaried`, this `Commission` is *not* a derived class. So, its `earnings` and `toString` member functions do not override base-class virtual functions, as they did in [Section 10.12](#).

[Click here to view code image](#)

```

1  // Fig. 10.36: Commission.h
2  // Commission compensation model.
3  #pragma once
4  #include <iostream>
5
6  class Commission {
7  public:
8      Commission(double grossSales, double commissionRate);
9      double earnings() const;
```

```
10     std::string toString() const;
11 private:
12     double m_grossSales{0.0};
13     double m_commissionRate{0.0};
14 }
```



Fig. 10.36 | Commission compensation model.

[Click here to view code image](#)

```
1 // Fig. 10.37: Commission.cpp
2 // Commission member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this w
5 #include "Commission.h" // class definition
6 using namespace std;
7
8 // constructor
9 Commission::Commission(double grossSales, d
10 : m_grossSales{grossSales}, m_commissionR
11
12     if (m_grossSales < 0.0) {
13         throw invalid_argument("Gross sales m
14     }
15
16     if (m_commissionRate <= 0.0 || m_commissi
17         throw invalid_argument("Commission ra
18     }
19 }
20
21 // calculate earnings
22 double Commission::earnings() const {
23     return m_grossSales * m_commissionRate;
24 }
25
26 // return string containing Commission info.
```

```
27     string Commission::toString() const {
28         return fmt::format("gross sales: ${:.2f}\n"
29                            m_grossSales, m_commissionRate);
30     }
```



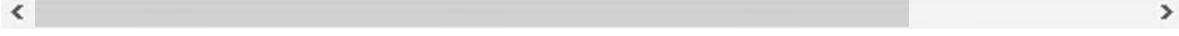
Fig. 10.37 | Commission member-function definitions. (Part 2 of 2.)

Employee Class Definition

As in [Section 10.12](#), each Employee ([Fig. 10.38](#)) has a compensation model (line 21). However, in this example, the compensation model is a `std::variant` object containing either a `Commission` or a `Salaried` object—note that this is an object, *not* a pointer to an object. Line 11's `using` declaration:

[Click here to view code image](#)

```
using CompensationModel = std::variant<Commission
```



defines the alias `CompensationModel` for the `std::variant` type:

[Click here to view code image](#)

```
std::variant<Commission, Salaried>
```

11 SE A Such using declarations (C++11) are known as **alias declarations** and are particularly useful when a `std::variant` type has many type parameters. At any given time, an object of our `std::variant` type can store either a `Commission` object or a `Salaried` object. We use our `CompensationModel` alias in line 21 to define the `std::variant` object that stores the Employee's compensation model.

[Click here to view code image](#)

```
1 // Fig. 10.38: Employee.h
2 // An Employee "has a" CompensationModel.
3 #pragma once // prevent multiple inclusions
```

```

4  #include <string>
5  #include <string_view>
6  #include <variant>
7  #include "Commission.h"
8  #include "Salaried.h"
9
10 // define a convenient name for the std::va
11 using CompensationModel = std::variant<Comm
12
13 class Employee {
14 public:
15     Employee(std::string_view name, Compensa
16     void setCompensationModel(CompensationMo
17     double earnings() const;
18     std::string toString() const;
19 private:
20     std::string m_name{};
21     CompensationModel m_model; // note this
22 };

```



Fig. 10.38 | An Employee “has a” CompensationModel. (Part 1 of 2.)

Type-Safe union

In C and C++, a **union**³⁶ is a region of memory that, over time, can contain objects of a variety of types. However, a union’s members share the same storage space, so a union may contain a maximum of one object at a time and requires enough memory to hold the largest of its members. A **std::variant object is often referred to as a type-safe union.**³⁷

36. Discussed in Section F.12.

37. “std::variant.” Accessed January 30, 2021. <https://en.cppreference.com/w/cpp/utility/variant>.

Employee Constructor and setCompensationModel Member Function

Class Employee's member functions perform the same tasks as in Fig. 10.28 with several modifications. The Employee class's constructor (lines 9–10) and setCompensationModel member function (lines 13–15) each receive a CompensationModel *object*. Unlike the composition-and-dependency-injection approach, the std::variant object **stores an actual object**, not a pointer to an object.

[Click here to view code image](#)

```
1 // Fig. 10.39: Employee.cpp
2 // Class Employee member-function definition
3 #include <string>
4 #include "fmt/format.h" // In C++20, this will be part of the standard library
5 #include "Employee.h"
6 using namespace std;
7
8 // constructor
9 Employee::Employee(string_view name, CompensationModel model)
10    : m_name{name}, m_model{model} {}
11
12 // change the Employee's CompensationModel
13 void Employee::setCompensationModel(CompensationModel model)
14 {
15     m_model = model;
16 }
17
18 // return the Employee's earnings
19 double Employee::earnings() const {
20     auto getEarnings = [] (const auto& model) {
21         return std::visit(getEarnings, m_model);
22     };
23
24     // return string representation of an Employee
25     string Employee::toString() const {
26         auto getString = [] (const auto& model) {
27             return fmt::format("{}\n{}", m_name, std::visit(getString, m_model));
28         };
29         return getString(m_model);
30     }
31 }
```





Fig. 10.39 | Class Employee member-function definitions. (Part 2 of 2.)

Employee earnings and `toString` Member Functions—Calling Member Functions with `std::visit`

A key difference between runtime polymorphism via class hierarchies and runtime polymorphism via `std::variant` is that a `std::variant` object cannot call member functions of the object it contains. Instead, you use the standard library function `std::visit` to invoke a function on the object stored in the `std::variant`. Consider lines 19–20 in member-function `earnings`:

[Click here to view code image](#)

```
auto getEarnings = [](const auto& model){return m_
return std::visit(getEarnings, m_model);
```



Line 19 defines the variable `getEarnings` and initializes it with a generic lambda expression (introduced in Fig. 6.14.2) that receives a reference to an object (`model`) and calls the object's `earnings` member function. This lambda expression can call `earnings` on *any object* with an `earnings` member function that takes no arguments and returns a value. Line 20 passes to function `std::visit` the `getEarnings` lambda expression and the `std::variant` object `m_model`. Function `std::visit` passes `m_model` to the lambda expression, then returns the result of calling `m_model`'s `earnings` member function.

Similarly, line 25 in `Employee`'s `toString` member function creates a lambda expression that returns the result of calling some object's `toString` member function. Line 26 calls `std::visit` to pass `m_model` to the lambda expression, which returns the result of calling `m_model`'s `toString` member function.

Testing Runtime Polymorphism with `std::variant` and `std::visit`

SE A In `main`, lines 12–13 create two `Employee` objects—the first

stores a `Salaried` object in its `std::variant` and the second stores a `Commission` object. Line 16 creates a vector of `Employees`, then lines 19–22 iterate through it calling each `Employee`'s `earnings` and `toString` member functions to demonstrate the runtime polymorphic processing, producing the same results as in [Section 10.12](#). **This ability to invoke common functionality on objects whose types are not related by a class hierarchy is often called **duck typing**:**

*“If it walks like a duck and it quacks like a duck, then it must be a duck.”*³⁸

38. “Duck Typing.” Accessed January 30, 2021. <https://w.wiki/Bin>.

That is, if an object has the appropriate member functions and its type is specified as a member of the `std::variant`, the object will work in this code.

[Click here to view code image](#)

```
1 // fig10_40.cpp
2 // Processing Employees with various compensation
3 #include <iostream>
4 #include <vector>
5 #include "fmt/format.h" // In C++20, this will be part of the standard library
6 #include "Employee.h"
7 #include "Salaried.h"
8 #include "Commission.h"
9 using namespace std;
10
11 int main() {
12     Employee salariedEmployee{"John Smith", 100000.0};
13     Employee commissionEmployee{"Sue Jones", 100000.0, 10000.0};
14
15     // create and initialize vector of three
16     vector<Employee> employees{salariedEmployee, commissionEmployee};
17
18     // print each Employee's information and
19     for (const Employee& employee : employees) {
20         cout << fmt::format("{}\nearned: ${}.\n", employee.toString(), employee.earnings());
```

```
21         employee.toString(), employe
22     }
23 }
```

```
< >
John Smith
salary: $800.00
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 10.40 | Processing Employees with various compensation models.

10.14 Multiple Inheritance

SE A So far, we've discussed single inheritance, in which each class is derived from exactly one base class. C++ also supports **multiple inheritance** —a class may inherit the members of two or more base classes. **Multiple inheritance is a complicated feature that should be used only by experienced programmers.** Some of the problems associated with multiple inheritance are so subtle that newer programming languages, such as Java and C#, support only single inheritance.³⁹ **Great care is required to design a system to use multiple inheritance properly. It should not be used when single inheritance and/or composition will do the job.**

SE A A common problem with multiple inheritance is that each base class might contain data members or member functions with the same name. This can lead to ambiguity problems when you attempt to compile. **The ISO C++ FAQ recommends doing multiple inheritance from only pure abstract base classes** to avoid this problem and others we'll discuss in this section and Section 10.14.1.⁴⁰

³⁹. More precisely, Java and C# only single *implementation* inheritance. They do allow multiple interface inheritance.

⁴⁰. “Inheritance — Multiple and Virtual Inheritance.” Accessed February 7, 2021.

<https://isocpp.org/wiki/faq/multiple-inheritance>.

Multiple-Inheritance Example

Let's consider a multiple-inheritance example using implementation inheritance (Figs. 10.41–10.45). Class `Base1` (Fig. 10.41) contains:

- one `private int` data member (`m_value`; line 11),
- a constructor (line 8) that sets `m_value`, and
- a `public` member function `getData` (line 9) that returns `m_value`.

[Click here to view code image](#)

```
1 // Fig. 10.41: Base1.h
2 // Definition of class Base1
3 #pragma once
4
5 // class Base1 definition
6 class Base1 {
7 public:
8     explicit Base1(int value) : m_value{value}
9     int getData() const {return m_value;}
10 private: // accessible to derived classes v
11     int m_value;
12 };
```



Fig. 10.41 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 10.42) is similar to class `Base1`, except that its `private` data is a `char` named `m_letter` (line 11). Like class `Base1`, `Base2` has a `public` member function `get-Data`, but this function returns `m_letter`'s value.

[Click here to view code image](#)

```
1 // Fig. 10.42: Base2.h
```

```
2 // Definition of class Base2
3 #pragma once
4
5 // class Base2 definition
6 class Base2 {
7 public:
8     explicit Base2(char letter) : m_letter{letter}
9     char getData() const {return m_letter;}
10    private: // accessible to derived classes via friend functions
11        char m_letter;
12    };

```



Fig. 10.42 | Demonstrating multiple inheritance—Base2.h.

Class Derived (Figs. 10.43–10.44) inherits from classes Basel and Base2 via multiple inheritance. Class Derived has:

- a private data member of type double named `m_real` (Fig. 10.43, line 19),
- a constructor to initialize all the data of class Derived,
- a public member function `getReal` that returns the value of `m_real`, and
- a public member function `toString` that returns a string representation of a Derived object.

[Click here to view code image](#)

```
1 // Fig. 10.43: Derived.h
2 // Definition of class Derived which inherits from two base classes
3 // (Basel and Base2).
4 #pragma once
5
6 #include <iostream>
7 #include <string>
8 #include "Basel.h"
```

```
9 #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2
14 public:
15     Derived(int value, char letter, double r
16     double getReal() const;
17     std::string toString() const;
18 private:
19     double m_real; // derived class's private
20 };
```



Fig. 10.43 | Demonstrating multiple inheritance—Derived.h.

[Click here to view code image](#)

```
1 // Fig. 10.44: Derived.cpp
2 // Member-function definitions for class De
3 #include "fmt/format.h" // In C++20, this w
4 #include "Derived.h"
5
6 // constructor for Derived calls Base1 and :
7 Derived::Derived(int value, char letter, do
8     : Base1{value}, Base2{letter}, m_real{re
9
10 // return real
11 double Derived::getReal() const {return m_r
12
13 // display all data members of Derived
14 string Derived::toString() const {
15     return fmt::format("int: {}; char: {}; d
16             Base1::getData(), Base2::getData()
17 }
```



Fig. 10.44 | Demonstrating multiple inheritance—Derived.cpp.

SE A For multiple inheritance (in Fig. 10.43), we follow the colon (:) after class Derived with a **comma-separated list of base classes** (line 13). In Fig. 10.44, notice that constructor Derived explicitly calls base-class constructors for each base class—Base1 and Base2—using the member-initializer syntax (line 8). **The base-class constructors are called in the order that the inheritance is specified. If the member-initializer list does not explicitly call a base class's constructor, the base class's default constructor will be called implicitly.**

Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes

Member function `toString` (lines 14–17) returns a string representation of a Derived object’s contents. It uses all of the Derived class’s `get` member functions. However, there’s an ambiguity problem. A **Derived object contains two `getData` functions**—one inherited from class Base1 and one inherited from class Base2. This problem is easy to solve by using the scope-resolution operator. `Base1::getData()` gets the value of the variable inherited from class Base1 (i.e., the `int` variable named `m_value`), and `Base2::getData()` gets the value of the variable inherited from class Base2 (i.e., the `char` variable named `m_letter`).

Testing the Multiple-Inheritance Hierarchy

Figure 10.45 tests the classes in Figs. 10.41–10.44. Line 11 creates Base1 object `base1` and initializes it to the `int` value 10. Line 12 creates Base2 object `base2` and initializes it to the `char` value 'Z'. Line 13 creates Derived object `derived` and initializes it to contain the `int` value 7, the `char` value 'A' and the `double` value 3.5.

[Click here to view code image](#)

```
1 // fig10_45.cpp
2 // Driver for multiple-inheritance example.
3 #include <iostream>
```

```
4 #include "fmt/format.h" // In C++20, this w
5 #include "Base1.h"
6 #include "Base2.h"
7 #include "Derived.h"
8 using namespace std;
9
10 int main() {
11     Base1 base1{10}; // create Base1 object
12     Base2 base2{'Z'}; // create Base2 object
13     Derived derived{7, 'A', 3.5}; // create Derived object
14
15     // print data in each object
16     cout << fmt::format("{}: {}\n{}: {}\n{}: {}",
17                         "Object base1 contains", base1.getData(),
18                         "Object base2 contains the character", base2.getData(),
19                         "Object derived contains", derived.getData());
20
21     // print data members of derived-class object
22     // scope resolution operator resolves getReal()
23     cout << fmt::format("{}\n{}: {}\n{}: {}\n{}: {}",
24                         "Data members of Derived can be treated as an object",
25                         "int", derived.Base1::getData(),
26                         "char", derived.Base2::getData(),
27                         "double", derived.getReal());
28
29     cout << "Derived can be treated as an object";
30
31     // treat Derived as a Base1 object
32     Base1* base1Ptr = &derived;
33     cout << fmt::format("base1Ptr->getData() = ", base1Ptr->getData());
34
35     // treat Derived as a Base2 object
36     Base2* base2Ptr = &derived;
37     cout << fmt::format("base2Ptr->getData() = ", base2Ptr->getData());
38
39 }
40 }
```



```
Object base1 contains: 10
Object base2 contains the character: Z
Object derived contains: int: 7; char: A; double: 3.5

Data members of Derived can be accessed individually:
int: 7
char: A
double: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 10.45 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 16–19 display each object’s data values. For objects `base1` and `base2`, we invoke each object’s `getData` member function. Even though there are *two* `getData` functions in this example, the calls are *not ambiguous*. In line 17, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`’s `getData` is called. In line 18, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`’s `getData` is called. Line 19 gets `derived`’s contents by calling its `toString` member function.

Lines 23–27 output `derived`’s contents again by using class `Derived`’s `get` member functions. Again, there is an *ambiguity* problem—this object contains `getData` functions from both class `Base1` and class `Base2`. The expression

```
derived.Base1::getData()
```

gets the value of `m_value` inherited from class `Base1` and

```
derived.Base2::getData()
```

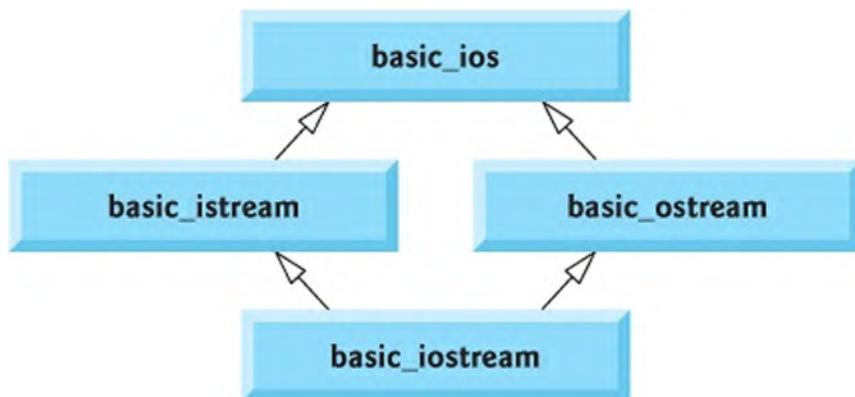
gets the value of `m_letter` inherited from class `Base2`.

Demonstrating the *Is-a* Relationships in Multiple Inheritance

The *is-a* relationships of *single inheritance* also apply in *multiple-inheritance* relationships. To demonstrate this, line 32 assigns derived's address to the `Base1` pointer `base1Ptr`. This is allowed because **a Derived object is a Base1 object**. Line 34 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 37 assigns derived's address to the `Base2` pointer `base2Ptr`. This is allowed because **a Derived object is a Base2 object**. Line 39 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

10.14.1 Diamond Inheritance

In Section 10.14, we discussed *multiple inheritance*, the process by which one class inherits from *two or more* classes. Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream`, as shown in the following diagram:



Class `basic_ios` is the base class of both `basic_istream` and `basic_ostream`. Each is formed with *single inheritance*. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables class `basic_iostream` objects to provide the functionality of `basic_istreams` and `basic_ostreams`. In multiple-inheritance hierarchies, the inheritance described in this diagram is referred to as **diamond inheritance**.

Classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, so a potential problem exists for `basic_iostream`. It could

inherit *two* copies of `basic_ios`'s members— one via `basic_istream` and one via `basic_ostream`. This would be *ambiguous* and would result in a compilation error—the compiler would not know which copy of `basic_ios`'s members to use. Let's see how **using virtual base classes solves this problem.**

Compilation Errors Produced When Ambiguity Arises in Diamond Inheritance

Figure 10.46 demonstrates the *ambiguity* that can occur in *diamond inheritance*. Class `Base` (lines 8–11) contains pure virtual function `print` (line 10). Classes `DerivedOne` (lines 14–18) and `DerivedTwo` (lines 21–25) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain a **base-class subobject**—i.e., the members of class `Base` in this example.

[Click here to view code image](#)

```
1 // fig10_46.cpp
2 // Attempting to polymorphically call a fun-
3 // inherited from each of two base classes.
4 #include <iostream>
5 using namespace std;
6
7 // class Base definition
8 class Base {
9 public:
10     virtual void print() const = 0; // pure
11 };
12
13 // class DerivedOne definition
14 class DerivedOne : public Base {
15 public:
16     // override print function
17     void print() const override {cout << "De
18 };
```

```

19 // class DerivedTwo definition
20 class DerivedTwo : public Base {
21 public:
22     // override print function
23     void print() const override {cout << "De
24     };
25
26
27 // class Multiple definition
28 class Multiple : public DerivedOne, public :
29 public:
30     // qualify which version of function pri
31     void print() const override {DerivedTwo:
32     };
33
34 int main() {
35     Multiple both{}; // instantiate a Multip
36     DerivedOne one{}; // instantiate a Derive
37     DerivedTwo two{}; // instantiate a Derive
38     Base* array[3]{}; // create array of bas
39
40     array[0] = &both; // ERROR--ambiguous
41     array[1] = &one;
42     array[2] = &two;
43
44     // polymorphically invoke print
45     for (int i{0}; i < 3; ++i) {
46         array[i] ->print();
47     }
48 }
```



Microsoft Visual C++ compiler error message:

```
c:\Users\PaulDeitel\Documents\examples\ch10\fig1
error C2594: '=': ambiguous conversions from 'Mu
```



Fig. 10.46 | Attempting to polymorphically call a function that is inherited from each of two base classes. (Part 2 of 2.)

Class `Multiple` (lines 28–32) inherits from *both* class `DerivedOne` and class `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 31). Notice that we must *qualify* the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 34–48) declares objects of classes `Multiple` (line 35), `DerivedOne` (line 36) and `DerivedTwo` (line 37). Line 38 declares an array of `Base*` pointers. Each array element is initialized with the address of an object (lines 40–42). An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[0]`. The object `both` actually contains two `Base` subobjects. The compiler does not know which subobject the pointer `array[0]` should point to, so it generates a compilation error indicating an *ambiguous conversion*.

10.14.2 Eliminating Duplicate Subobjects with `virtual` Base-Class Inheritance

 **The problem of *duplicate subobjects* is resolved with `virtual inheritance`.** When a base class is inherited as `virtual`, only *one* subobject will appear in the derived class. Figure 10.47 revises the program of Fig. 10.46 to use a `virtual` base class.

[Click here to view code image](#)

```
1 // fig10_47.cpp
2 // Using virtual base classes.
3 #include <iostream>
4 using namespace std;
5
6 // class Base definition
7 class Base {
8 public:
```

```
9         virtual void print() const = 0; // pure
10    };
11
12 // class DerivedOne definition
13 class DerivedOne : virtual public Base {
14 public:
15     // override print function
16     void print() const override {cout << "De
17    };
18
19 // class DerivedTwo definition
20 class DerivedTwo : virtual public Base {
21 public:
22     // override print function
23     void print() const override {cout << "De
24    };
25
26 // class Multiple definition
27 class Multiple : public DerivedOne, public Base {
28 public:
29     // qualify which version of function pri
30     void print() const override {DerivedTwo::
31    };
32
33 int main() {
34     Multiple both; // instantiate Multiple ob
35     DerivedOne one; // instantiate DerivedOne
36     DerivedTwo two; // instantiate DerivedTw
37     Base* array[3];
38
39     array[0] = &both; // allowed now
40     array[1] = &one;
41     array[2] = &two;
42
43     // polymorphically invoke function print
44     for (int i = 0; i < 3; ++i) {
45         array[i]->print();
```

```
46      }
47 }
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 10.47 | Using virtual base classes. (Part 2 of 2.)

The key change is that classes DerivedOne (line 13) and DerivedTwo (line 20) each inherit from Base using `virtual public Base`. Since both classes inherit from Base, they each contain a Base subobject. The benefit of virtual inheritance is not apparent until class Multiple inherits from DerivedOne and DerivedTwo (line 27). **Since each base class used virtual inheritance, the compiler ensures that Multiple inherits only one Base subobject.** This eliminates the ambiguity error generated by the compiler in Fig. 10.46. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[0]` in line 39 in main. The `for` statement in lines 44–46 polymorphically calls `print` for each object.

Constructors in Multiple-Inheritance Hierarchies with `virtual` Base Classes

SE  Implementing hierarchies with virtual base classes is simpler if *default constructors* are used for the base classes. Figures 10.46 and 10.47 use compiler-generated *default constructors*. If a virtual base class provides a constructor that requires arguments, the derived-class implementations become more complicated. **The most derived class must explicitly invoke the virtual base class's constructor.** For this reason, consider providing a default constructor for virtual base classes, so the derived classes do not need to explicitly invoke the virtual base class's constructor.^{41,42}

^{41,42} “Inheritance — Multiple and Virtual Inheritance — What special considerations do I need to know about when I use virtual inheritance?” Accessed February 7, 2021. <https://isocpp.org/wiki/faq/multiple-inheritance#virtual->

[inheritance-abcs](#).

42. “Inheritance — Multiple and Virtual Inheritance — What special considerations do I need to know about when I inherit from a class that uses virtual inheritance?” Accessed February 7, 2021. <https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-ctors>.

10.15 protected Class Members

Chapter 9 introduced the access specifiers `public` and `private`. A base class’s `public` members are accessible within its class and anywhere that the program has access to an object of that class or one of its derived classes. A base class’s `private` members are accessible only within its body and to its `friends`. The access specifier `protected` offers an intermediate level of protection between `public` and `private` access. Such members are accessible within that base class, by members and `friends` of that base class, and by members and `friends` of any classes derived from that base class.

In `public` inheritance, all `public` and `protected` base-class members retain their original access when they become members of the derived class:

- `public` base-class members become `public` derived-class members, and
- `protected` base-class members become `protected` derived-class members.

A base class’s `private` members are *not accessible outside the class itself*. Derived classes can access a base class’s `private` members only through the `public` or `protected` member functions inherited from the base class. Derived-class member functions can refer to `public` and `protected` members inherited from the base class by their member names.

Problems with `protected` Data

 It’s best to avoid `protected` data members because they create some serious problems:

- A derived-class object does not have to use a member function to set a base-class `protected` data member’s value. So, an invalid value can be assigned, leaving the object in an inconsistent state. For

example, if `CommissionEmployee`'s data member `m_grossSales` is protected (and the class is not `final`), a derived-class object can assign a negative value to `m_grossSales`.

-  **Derived-class member functions are more likely to be written so that they depend on the base class's data.** Derived classes should depend only on the base-class non-private member functions. If we were to change the name of a base-class protected data member, we'd need to modify every derived class that references the data directly. Such software is said to be **fragile** or **brittle**. **A small change in the base class can “break” the derived class. This is known as the fragile base-class problem⁴³** and is a key reason why the C++ Core Guidelines recommend avoiding **protected data**.⁴⁴ You should be able to make changes in a base class without having to modify its derived classes.

43. “Fragile base class.” Accessed February 7, 2021. <https://w.wiki/yFZ>.

44. “C++ Core Guidelines — C.133: Avoid protected data.” Accessed February 7, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-protected>.

 In most cases, it's better to use private data members to encourage proper software engineering. **Declaring base-class data members `private` allows you to change the base-class implementation without having to change derived-class implementations. This makes your code easier to maintain, modify and debug.**

 A derived class can change the state of private base-class data members only through non-private base-class member functions inherited into the derived class. **If a derived class could access its base class's `private` data members, classes that inherit from that derived class could access the data members as well**, and the benefits of information hiding would be lost.

protected Base-Class Member Functions

 One use of `protected` is to **define base-class member functions that should not be exposed to client code but should be accessible in**

derived classes. A use-case for this is to enable a derived class to override the base-class protected virtual function and also call the original base-class version from the derived class's implementation.⁴⁵ We used this capability in the example of [Section 10.11](#), Non-Virtual Interface (NVI) Idiom.

45. Herb Sutter, “Virtuality.” Accessed February 3, 2021.
<http://www.gotw.ca/publications/mill18.htm>.

10.16 public, protected and private Inheritance

You can choose between **public**, **protected** or **private** inheritance. **public** inheritance is the most common, and **protected** inheritance is rare. The following diagram summarizes for each inheritance type the accessibility of base-class members in a derived class. The first column contains the base-class member access specifiers.

Base-class access specifier	public inheritance	protected inheritance	private inheritance
public	<p>public in derived class.</p> <p>Can be accessed directly by member functions, friend functions and nonmember functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
protected	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
private	<p>Inaccessible in the derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions inherited from the base class.</p>	<p>Inaccessible in the derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions inherited from the base class.</p>	<p>Inaccessible in the derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions inherited from the base class.</p>

When deriving a class with `public` inheritance:

- `public` base-class members become `public` derived-class members, and
- `protected` base-class members become `protected` derived-class members.

A base class's `private` members are *never* accessible directly in the derived class but can be accessed by calling inherited `public` and `protected` base-class member functions designed to access those

private members.

When deriving a class with protected inheritance, public and protected base-class members become protected derived-class members. When deriving a class with private inheritance, public and protected base-class members become private derived-class members.

SE A Classes created with **private** and **protected** inheritance do not have *is-a* relationships with their base classes because the base class's **public** members are not accessible to the derived class's client code.

Default Inheritance in **classes** vs. **structs**

Both **class** and **struct** (defined in [Section 9.21](#)) can be used to define new types. However, there are two key differences between a **class** definition and a **struct** definition. The first is that, by default, **class** members are **private** and **struct** members are **public**. The other difference is in the default inheritance type. A **class** definition like

```
class Derived : Base {  
    // ...  
};
```

uses **private** inheritance by default, whereas a **struct** definition like

```
struct Derived : Base {  
    // ...  
};
```

uses **public** inheritance by default.

10.17 Wrap-Up

This chapter continued our object-oriented programming (OOP) discussion with introductions to inheritance and runtime polymorphism. You created derived classes that inherited base classes' capabilities, then customized or enhanced them. We distinguished between the *has-a* composition relationship and the *is-a* inheritance relationship.

We explained and demonstrated runtime polymorphism with inheritance

hierarchies. You wrote programs that processed objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class. You manipulated those objects via base-class pointers and references.

We explained that runtime polymorphism helps you design and implement systems that are easier to extend, enabling you to add new classes with little or no modification to the program's general portions. We used a detailed illustration to help you understand how runtime polymorphism, virtual functions and dynamic binding can be implemented "under the hood."

We introduced the non-virtual interface idiom (NVI) in which each base-class function serves only one purpose—as either a public non-virtual function that client code calls to perform a task or as a private (or protected) virtual function that derived classes can customize. You saw that when using this idiom, the virtual functions are internal implementation details hidden from the client code, making systems easier to maintain and evolve.

We initially focused on implementation inheritance, then pointed out that decades of experience with real-world, business-critical and mission-critical systems has shown that it can be difficult to maintain and modify such systems. So, we refactored our Employee–payroll example to use interface inheritance in which the base class contained only pure virtual functions that derived concrete classes were required to implement. In that example, we introduced the composition-and-dependency-injection approach in which a class contains a pointer to an object that provides behaviors required by objects of the class. Then, we discussed how this interface-based approach makes the example easier to modify and evolve.

Next, we looked at another way to implement runtime polymorphism by processing objects of classes not related by inheritance, using duck typing via C++17's class template `std::variant` and the standard-library function `std::visit`.

We demonstrated multiple inheritance, discussed its potential problems and showed how virtual inheritance can be used to solve them. We introduced the protected access specifier—derived-class member functions and friends of the derived class can access protected base-class members. We also explained the three types of inheritance—public, protected

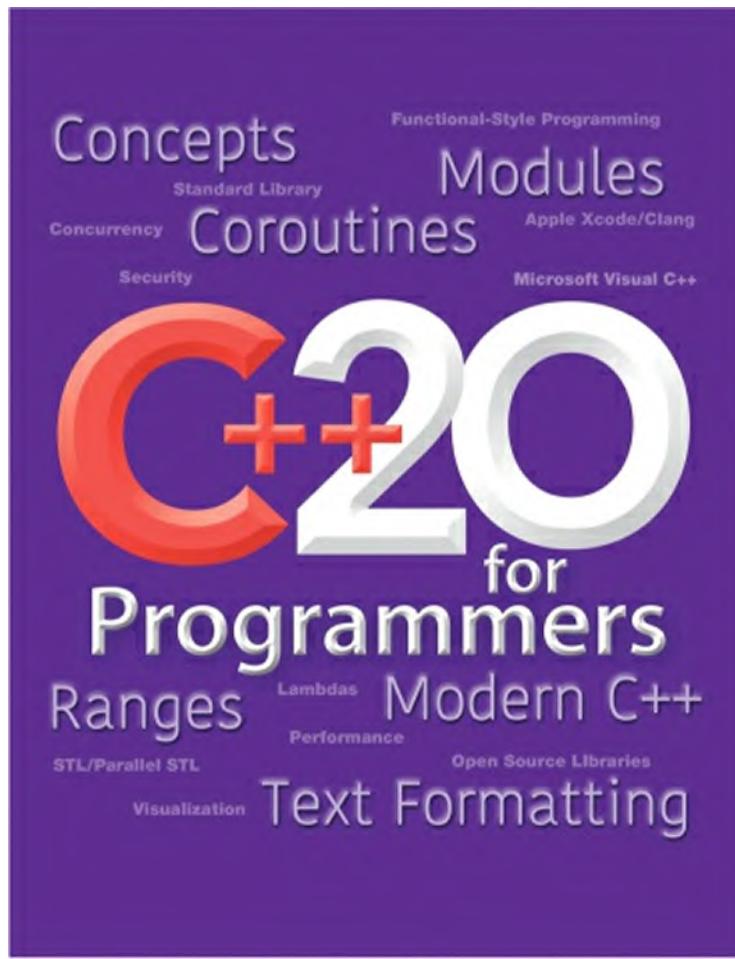
and `private`—and the accessibility of base-class members in a derived class when using each type.

Finally, we overviewed other runtime-polymorphism techniques and several template-based compile-time polymorphism approaches. We'll implement some of these in [Chapter 15](#). There you'll see that C++20's new concepts capabilities obviate some older techniques and strengthen your “toolkit” for creating compile-time, template-based solutions. We also provided links to advanced resources for developers who wish to dig deeper.

A key goal of this chapter was to familiarize you with the mechanics of inheritance and runtime polymorphism with `virtual` functions. Now, you'll be able to better appreciate newer polymorphism idioms and techniques that can promote ease of modifiability and better performance. We cover some of these modern idioms in later chapters.

In [Chapter 11](#), we continue our object-oriented programming presentation with operator overloading, which enables existing operators to work with custom class-type objects as well. For example, you'll see how to overload the `<<` operator to output a complete array without explicitly using an iteration statement. You'll use “smart pointers” to manage dynamically allocated memory and ensure that it's released when no longer needed. We'll introduce the remaining special member functions and discuss rules and guidelines for using them. We'll consider move semantics, which enable object resources (such as dynamically allocated memory) to move from one object to another when an object is going out of scope. As you'll see, this can save memory and increase performance.

Chapter 11. Operator Overloading, Copy/Move Semantics and Smart Pointers



Objectives

In this chapter, you'll:

- Use built-in `string` class overloaded operators.
- Use operator overloading to help you craft valuable classes.
- Understand the special member functions and when to implement them for custom types.
- Understand when objects should be moved vs. copied.

- Use *rvalue* references and move semantics to eliminate unnecessary copies of temporary objects that are going out of scope, improving program performance.
 - Understand why you should avoid dynamic memory management with operators `new` and `delete`.
 - Manage dynamic memory automatically with smart pointers.
 - Craft a polished `MyArray` class that defines the five special member functions to support copy and move semantics, and overloads many unary and binary operators.
 - Use C++20's three-way comparison operator (`<=>`).
 - Convert objects to other types.
 - Use keyword `explicit` to prevent constructors and conversion operators from being used for implicit conversions.
 - Experience a “light-bulb moment” when you'll truly appreciate the elegance and beauty of the class concept.
-

Outline

- 11.1 Introduction**
- 11.2 Using the Overloaded Operators of Standard Library Class `string`**
- 11.3 Operator Overloading Fundamentals**
 - 11.3.1 Operator Overloading Is Not Automatic
 - 11.3.2 Operators That Cannot Be Overloaded
 - 11.3.3 Operators That You Do Not Have to Overload
 - 11.3.4 Rules and Restrictions on Operator Overloading
- 11.4 (Downplaying) Dynamic Memory Management with `new` and `delete`**
- 11.5 Modern C++ Dynamic Memory Management—RAII and Smart Pointers**
 - 11.5.1 Smart Pointers
 - 11.5.2 Demonstrating `unique_ptr`

11.5.3 `unique_ptr` Ownership

11.5.4 `unique_ptr` to a Built-In Array

11.6 MyArray Case Study: Crafting a Valuable Class with Operator Overloading

11.6.1 Special Member Functions

11.6.2 Using Class `MyArray`

11.6.3 `MyArray` Class Definition

11.6.4 Constructor That Specifies a `MyArray`'s Size

11.6.5 C++11 Passing a Braced Initializer to a Constructor

11.6.6 Copy Constructor and Copy Assignment Operator

11.6.7 Move Constructor and Move Assignment Operator

11.6.8 Destructor

11.6.9 `toString` and `size` Functions

11.6.10 Overloading the Equality (`==`) and Inequality (`!=`) Operators

11.6.11 Overloading the Subscript (`[]`) Operator

11.6.12 Overloading the Unary `bool` Conversion Operator

11.6.13 Overloading the Preincrement Operator

11.6.14 Overloading the Postincrement Operator

11.6.15 Overloading the Addition Assignment Operator (`+=`)

11.6.16 Overloading the Binary Stream Extraction (`>>`) and Stream Insertion (`<<`) Operators

11.6.17 `friend` Function `swap`

11.7 C++20 Three-Way Comparison Operator (`<=>`)

11.8 Converting Between Types

11.9 `explicit` Constructors and Conversion Operators

11.10 Overloading the Function Call Operator ()

11.11 Wrap-Up

11.1 Introduction

This chapter shows how to enable C++'s existing operators to work with class objects—a process called **operator overloading**. One example of an overloaded operator in standard C++ is `<<`, which is used *both* as

- the stream insertion operator *and*
- the bitwise left-shift operator (which is discussed in Appendix E).

Similarly, `>>` also is overloaded; it's used both as

- the stream extraction operator *and*
- the bitwise right-shift operator (which is discussed in Appendix E).

You've actually been using overloaded operators since early in the book. Various overloads are built into the core C++ language itself. For example, C++ overloads the addition operator (`+`) to perform differently, based on its context in integer, floating-point and pointer arithmetic with data of fundamental types.

You can overload most operators to be used with class objects. The compiler generates the appropriate code based on the operand types. The jobs performed by overloaded operators also can be performed by explicit function calls, but operator notation is often more natural.

string Class Overloaded Operators Demonstration

In [Section 2.7](#)'s objects-natural case study, we introduced the standard library's `string` class and demonstrated a few of its features, such as string concatenation with the `+` operator. Here, we demonstrate many additional `string`-class overloaded operators, so you can appreciate how valuable operator overloading is in a key standard library class before implementing it in your own custom classes. Next, we'll present operator-overloading fundamentals.

Dynamic Memory Management and Smart Pointers

We introduce dynamic memory management, which enables a program to acquire additional memory it needs for objects at runtime rather than at compile-time and release that memory when it's no longer needed so it can be used for other purposes. We discuss the potential problems with dynamically allocated memory, such as forgetting to release memory that's no longer needed—known as a **memory leak**. Then, we introduce smart pointers, which can automatically release dynamically allocated memory for you. As you'll see, **smart pointers**, when coupled with the **RAII (Resource Acquisition Is Initialization) strategy**, enable you to eliminate subtle memory leak issues.

MyArray Case Study

Next, we present one of the book’s capstone case studies. We build a custom `MyArray` class that uses overloaded operators and other capabilities to solve various problems with C++’s native pointer-based arrays. We introduce and implement the five **special member functions** you can define in each class—the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. Sometimes the default constructor also is included as a special member function. We introduce copy semantics and move semantics, which help tell a compiler when it can move resources from one object to another to avoid costly unnecessary copies.

`MyArray` uses smart pointers and RAII, and overloads many unary and binary operators, including

- `=` (assignment),
- `==` (equality),
- `!=` (inequality),
- `[]` (subscript),
- `++` (increment),
- `+=` (addition assignment),
- `>>` (stream extraction) and
- `<<` (stream insertion).

We also show how to define a conversion operator that converts a `MyArray` to a `true` or `false` `bool` value, indicating whether a `MyArray` contains elements or is empty.

Many of our readers have said that working through the `MyArray` case study is a “light bulb moment,” helping them truly appreciate what classes and object technology are all about. Once you master this `MyArray` class, you’ll indeed understand the essence of object technology—crafting, using and reusing valuable classes—and sharing them with colleagues and perhaps the entire C++ open-source community.

C++20’s Three-Way Comparison Operator (`<= >`)

20 We introduce C++20’s new three-way comparison operator (`<= >`), which

is also referred to as the “spaceship operator.”¹

1. “Spaceship operator” was coined by Randal L. Schwartz when he was teaching the same operator in a Perl programming course—the operator reminded him of a spaceship in an early video game.
<https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftItp14/m/jcpli=1>.

Conversion Operators

We discuss overloaded conversion operators and conversion constructors in more depth. In particular, we demonstrate how implicit conversions can cause subtle problems. Then, we use `explicit` to prevent those problems.

11.2 Using the Overloaded Operators of Standard Library Class `string`

Chapter 8 presented class `string` in detail. Figure 11.1 demonstrates many of class `string`’s overloaded operators and several other useful member functions, including `empty`, `substr` and `at`:

- `empty` determines whether a `string` is empty,
- `substr` (for “substring”) returns a `string` that’s a portion of an existing `string`, and
- `at` returns the character at a specific index in a `string` (after checking that the index is in range).

For discussion purposes, we split this example into small chunks of code, each followed by its output.

[Click here to view code image](#)

```
1 // fig11_01.cpp
2 // Standard library string class test program
3 #include <iostream>
4 #include <string>
5 #include <string_view>
6 #include "fmt/format.h" // in C++20, this will be <format.h>
7 using namespace std;
8
```

```
9    int main() {
```

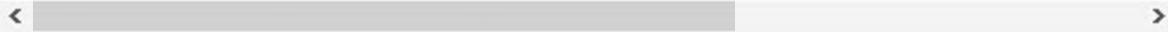


Fig. 11.1 | Standard library string class test program.

Creating `string` and `string_view` Objects and Displaying Them with `cout` and Operator `<<`

Lines 10–13 create three `strings` and a `string_view`:

- the `string` `s1` is initialized with the literal "happy",
- the `string` `s2` is initialized with the literal " birthday",
- the `string` `s3` uses `string`'s default constructor to create an empty string, and
- the `string_view` `v` is initialized to reference the characters in the literal "hello".

Lines 16–17 output these three objects, using `cout` and operator `<<`, which the `string` and `string_view` class designers overloaded to handle `string` objects.

[Click here to view code image](#)

```
10   string s1{"happy"}; // initialize string from literal
11   string s2{" birthday"}; // initialize string from literal
12   string s3; // creates an empty string
13   string_view v{"hello"}; // initialize string_view from literal
14
15   // output strings and string_view
16   cout << fmt::format("s1: \"{}\"; s2: \"{}\";
17                      s3: \"\"; v: \"{}\"", s1, s2, s3, v);
18
```

```
s1: "happy"; s2: " birthday"; s3: ""; v: "hello"
```

Comparing string Objects with the Equality and Relational Operators

20 Lines 20–26 show the results of comparing `s2` to `s1` by using class `string`'s overloaded equality and relational operators. These perform lexicographical comparisons (that is, like a dictionary ordering) using the numerical values of the characters in each `string` (see Appendix B, ASCII Character Set). When you use C++20's `format` function to convert a `bool` to a `string`, `format` produces the `string` "true" or the `string` "false".

[Click here to view code image](#)

```
19 // test overloaded equality and relational
20 cout << "The results of comparing s2 and s1
21     << fmt::format("s2 == s1: {}\\n", s2 == s1)
22     << fmt::format("s2 != s1: {}\\n", s2 != s1)
23     << fmt::format("s2 > s1: {}\\n", s2 > s1)
24     << fmt::format("s2 < s1: {}\\n", s2 < s1)
25     << fmt::format("s2 >= s1: {}\\n", s2 >= s1)
26     << fmt::format("s2 <= s1: {}\\n\\n", s2 <= s1)
```



```
The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true
s2 > s1: false
s2 < s1: true
s2 >= s1: false
s2 <= s1: true
```

string Member Function `empty`

Line 31 uses `string` member function `empty`, which returns `true` if the `string` is empty; otherwise, it returns `false`. The object `s3` was initialized with the default constructor, so it is indeed empty.

[Click here to view code image](#)

```
28 // test string member function empty
29 cout << "Testing s3.empty():\n";
30
31 if (s3.empty()) {
32     cout << "s3 is empty; assigning s1 to s3
33     s3 = s1; // assign s1 to s3
34     cout << fmt::format("s3 is \"{}\"\n\n",
35 }
36
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

string Copy Assignment Operator

Line 33 demonstrates class `string`'s **overloaded copy assignment operator** by assigning `s1` to `s3`. Line 34 outputs `s3` to demonstrate that the assignment worked correctly.

string Concatenation and C++14 string-object Literals

Line 38 demonstrates class `string`'s overloaded `+=` operator for **string concatenation assignment**. In this case, the contents of `s2` are appended to `s1`, thus modifying its value. Then line 39 outputs the resulting string that's stored in `s1`. Line 42 demonstrates that you also may append a C-string literal to a `string` object by using operator `+=`. Line 43 displays the result.

[Click here to view code image](#)

```
37 // test overloaded string concatenation ass
38 s1 += s2; // test overloaded concatenation
39 cout << fmt::format("s1 += s2 yields s1 = {
40
```

```
41 // test string concatenation with a C string
42 s1 += " to you";
43 cout << fmt::format("s1 += \" to you\" yields\n");
44
45 // test string concatenation with a C++14 string-object literal
46 s1 += ", have a great day!"s; // s after "
47 cout << fmt::format(
48         "s1 += \"", have a great day!"s);
49
```

```
s1 += s2 yields s1 = happy birthday
s1 += " to you" yields s1 = happy birthday to yo
s1 += ", have a great day!"s yields
s1 = happy birthday to you, have a great day!
```

14 Similarly, line 46 concatenates `s1` with a C++14 **string-object literal**, which is indicated by placing the letter `s` immediately following the closing `"` of a string literal, as in

```
", have a great day!"s
```

The preceding literal actually results in a call to a C++ standard library function that returns a `string` object containing the literal's characters. Lines 47–48 display the new value of `s1`.

string Member Function **substr**

Class `string` provides member function **substr** (lines 53 and 58) to return a `string` containing a portion of the `string` object on which the function is called. The call to `substr` in line 53 obtains a 14-character substring of `s1` starting at position 0. The call in line 58 obtains a substring starting from position 15 of `s1`. When the second argument is not specified, `substr` returns the remainder of the `string` on which it's called.

[Click here to view code image](#)

```
50 // test string member function substr
51 cout << fmt::format("{} {}\n{}\n\n",
52                     "The substring of s1 starting at
53                     14 characters, s1.substr(0, 14)
54
55 // test substr "to-end-of-string" option
56 cout << fmt::format("{} {}\n{}\n\n",
57                     "The substring of s1 starting at
58                     location 15, s1.substr(15), is:
59
```

< >

```
The substring of s1 starting at location 0 for 1
14), is:
happy birthday
```

```
The substring of s1 starting at location 15, s1.
to you, have a great day!
```

< >

string Copy Constructor

Line 61 creates string object `s4`, initializing it with a copy of `s1`. This calls class `string`'s **copy constructor**, which copies the contents of `s1` into the new object `s4`. You'll see how to define a custom copy constructor for your own class in [Section 11.6](#).

[Click here to view code image](#)

```
60 // test copy constructor
61 string s4{s1};
62 cout << fmt::format("s4 = {}\n\n", s4);
63
```

< >

```
s4 = happy birthday to you, have a great day!
```

Testing Self-Assignment with the `string` Copy Assignment Operator

Line 66 uses class `string`'s overloaded copy assignment (`=`) operator to demonstrate that it handles **self-assignment** properly, so `s4` still has the same value after the self-assignment. We'll see when we build class `MyArray` later in the chapter that **self-assignment must be handled carefully for objects that manage their own memory**, and we'll show how to deal with the issues.

[Click here to view code image](#)

```
64 // test overloaded copy assignment (=) oper.
65 cout << "assigning s4 to s4\n";
66 s4 = s4;
67 cout << fmt::format("s4 = {}\n\n", s4);
68
```



```
assigning s4 to s4
s4 = happy birthday to you, have a great day!
```

Initializing a `string` with a `string_view`

Line 71 demonstrates class `string`'s constructor that receives a `string_view`, in this case, copying the character data represented by the `string_view` `v` (line 13) into the new `string` `s5`.

[Click here to view code image](#)

```
69 // test string's string_view constructor
70 cout << "initializing s5 with string_view v
71 string s5{v};
72 cout << fmt::format("s5 is {}\n\n", s5);
```

73

```
< >
initializing s5 with string_view v
s5 is hello
```

string's [] Operator

Lines 75–76 use `string`'s overloaded `[]` operator in assignments to create *lvalues* for replacing characters in `s1`. Lines 77–78 output `s1`'s new value. The `[]` operator returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the expression appears. For example:

- If `[]` is used on a `non-const string`, the function returns a modifiable *lvalue*, which can be used on the left of an assignment (`=`) operator to assign a new value to that location in the `string`, as line lines 76–76.
- If `[]` is used on a `const string`, the function returns a nonmodifiable *lvalue* that can be used to obtain, but not modify, the value at that location in the `string`.

The overloaded `[]` operator does not perform bounds checking. So, you must ensure that operations using this operator do not accidentally manipulate elements outside the `string`'s bounds.

[Click here to view code image](#)

```
74 // test using overloaded subscript operator
75 s1[0] = 'H';
76 s1[6] = 'B';
77 cout << fmt::format("{}:{}\n{}",
78                     "after s1[0] = 'H' and s1[6] = '"
79
```



```
after s1[0] = 'H' and s1[6] = 'B', s1 is:  
Happy Birthday to you, have a great day!
```

string's at Member Function

Class `string` provides bounds checking in its member function `at`, which throws an exception if its argument is an invalid index. If the index is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the call appears. Line 83 demonstrates a call to function `at` with an invalid index causing an `out_of_range` exception. The error message in this case was produced by GNU C++.

[Click here to view code image](#)

```
80      // test index out of range with string m
81      try {
82          cout << "Attempt to assign 'd' to s1..
83          s1.at(100) = 'd'; // ERROR: subscript
84      }
85      catch (const out_of_range& ex) {
86          cout << fmt::format("An exception occ
87      }
88 }
```



```
Attempt to assign 'd' to s1.at(100) yields:  
An exception occurred: basic_string::at: __n (wh  
(which is 40)
```

11.3 Operator Overloading Fundamentals

 As you saw in Fig. 11.1, standard library class `string`'s overloaded operators provide a concise notation for manipulating `string`

objects. You can use operators with your own user-defined types as well. C++ allows most existing operators to be overloaded by defining **operator functions**. Once you define an operator function for a given operator and your custom class, that operator has meaning appropriate for objects of your class.

11.3.1 Operator Overloading Is Not Automatic

 You must write operator functions that perform the desired operations. An operator is overloaded by writing a non-static member function definition or a non-member function definition. An operator function's name is the keyword **operator**, followed by the symbol of the operator being overloaded. For example, the function name `operator+` would overload the addition operator (+). **When operators are overloaded as member functions, they must be non-static. They are called on an object of the class and operate on that object.**

11.3.2 Operators That Cannot Be Overloaded

Most of C++'s operators can be overloaded—the following operators cannot:²

2. Although it's possible to overload the address (&), comma (,), && and || operators, you should avoid doing so to avoid subtle errors. See <https://isocpp.org/wiki/faq/operator-overloading>. Accessed February 25, 2021.
- . (dot) member-selection operator
- . * pointer-to-member operator (discussed in Section 19.6)
- :: scope-resolution operator
- ?: conditional operator

11.3.3 Operators That You Do Not Have to Overload

Three operators work with objects of each new class by default:

-  **11** The **assignment operator** (=) may be used with *most* classes to perform **member-wise assignment** of the data members. The default assignment operator assigns each data member from the “source” object (on the right) to the “target” object (on the left). As you'll see

[Section 11.6.6](#), this can be dangerous for classes that have pointer members. So, you'll either explicitly overload the assignment operator or explicitly disallow the compiler from defining the default assignment operator. This is also true for the **C++11 move assignment operator**, which we discuss in [Section 11.6](#).

- The **address (&)** operator returns a pointer to the object.
- The **comma operator** evaluates the expression to its left then the expression to its right, and returns the latter expression's value. Though this operator can be overloaded, generally, it is not.³

3. “Operator Overloading.” Accessed February 24, 2021. <https://isocpp.org/wiki/faq/operator-overloading>.

11.3.4 Rules and Restrictions on Operator Overloading

As you prepare to overload operators for your own classes, there are several rules and restrictions you should keep in mind:

- **An operator's precedence cannot be changed by overloading.** Parentheses can be used to *force* the order of evaluation of overloaded operators in an expression.
- **An operator's grouping cannot be changed by overloading.** If an operator normally groups left-to-right, then so do its overloaded versions.
- **An operator's “arity” (the number of operands an operator takes) cannot be changed by overloading.** Overloaded unary operators remain unary operators and overloaded binary operators remain binary operators. The only ternary operator (?:) cannot be overloaded. Operators &, *, + and - each have unary and binary versions that can be overloaded separately.
- **Only existing operators can be overloaded.** You cannot create new ones.
- **You cannot overload operators to change how an operator works on only fundamental-type values.** For example, you cannot make the + operator subtract two ints.
- **Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.**

- **20 Related operators, like + and +=, generally must be overloaded separately.** In C++20, if you define == for your class, C++ provides != for you—it simply returns the opposite of ==.
- **When overloading (), [], -> or =, the operator overloading function must be declared as a class member.** You'll see later that this is required when the left operand must be an object of your custom class type. Operator functions for all other overloadable operators may be member functions or non-member functions.

SE A SE A You should overload operators for class types to work as closely as possible to how they work with fundamental types. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

11.4 (Downplaying) Dynamic Memory Management with `new` and `delete`

CG You can **allocate** and **deallocate** memory in a program for objects and for arrays of any built-in or user-defined type. This is known as **dynamic memory management**. In [Chapter 7](#), we introduced pointers and showed various old-style techniques you're likely to see in legacy code, then we showed improved Modern C++ techniques. We do the same here. For decades, C++ dynamic memory management was performed with operators **`new`** and **`delete`**. The C++ Core Guidelines recommend against using these operators directly.^{4,5} You'll likely see them in legacy C++ code, so we discuss them here. [Section 11.5](#) discusses Modern C++ dynamic memory management techniques, which we'll use in [Section 11.6](#)'s MyArray case study.

4. “R.11: Avoid calling `new` and `delete` explicitly.” Accessed February 26, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-newdelete>.
5. Operators `new` and `delete` can be overloaded, but this is beyond the scope of the book. If you do overload `new`, then you should overload `delete` in the same scope to avoid subtle dynamic memory management errors. Overloading `new` and `delete` is typically done for precise control over how memory is allocated and deallocated, often for performance. This might be used, for example, to preallocate a pool of memory, then create new objects within that pool to reduce

runtime memory-allocation overhead. For an overview of the so-called placement `new` and `delete` operators, see https://en.wikipedia.org/wiki/Placement_syntax.

The Old Way—Operators `new` and `delete`

You can use the `new` operator to dynamically reserve (that is, allocate) the exact amount of memory required to hold an object or built-in array. The object or built-in array is created on the **free store**—a region of memory assigned to each program for storing dynamically allocated objects. Once memory is allocated, you can access it via the pointer returned by operator `new`. When you no longer need the memory, you can return it to the free store by using the `delete` operator to deallocate (i.e., release) the memory, which can then be reused by future `new` operations.

Obtaining Dynamic Memory with `new`

Consider the following statement:

[Click here to view code image](#)

```
Time* timePtr{new Time{}};
```

 The `new` operator allocates storage of the proper size for a `Time` object, calls a default constructor to initialize the object (in this case, `Time`'s default constructor) and returns a pointer of the type specified to the right of the `new` operator (in this case, a `Time*`). In the preceding statement, class `Time`'s default constructor is called, because we did not supply arguments to initialize the `Time` object. If `new` is unable to find sufficient space in memory for the object, it throws a **`bad_alloc` exception**. Chapter 12 shows how to deal with `new` failures.

Releasing Dynamic Memory with `delete`

To destroy a dynamically allocated object and free the space for the object, use the `delete` operator as follows:

```
delete timePtr;
```

 This statement calls the destructor for the object to which `timePtr` points, then deallocates the object's memory, returning it to the free store. Not releasing dynamically allocated memory when it's no longer needed can

cause memory leaks that eventually lead to a system running out of memory prematurely. The problem might be even worse. If the leaked memory contains objects that manage other resources, those objects' destructors will not be called to release the resources, causing additional leaks.

 Do not delete memory that was not allocated by new. Doing so results in undefined behavior. After you delete a block of dynamically allocated memory, be sure not to delete the same block again, which typically causes a program to crash. One way to guard against this is to immediately set the pointer to nullptr—deleting such a pointer has no effect.

Initializing Dynamically Allocated Objects

You can initialize a newly allocated object with constructor arguments to the right of the type, as in:

[Click here to view code image](#)

```
Time* timePtr{new Time{12, 45, 0}};
```

which initializes a new Time object to 12:45:00 PM and assigns its pointer to timePtr.

Dynamically Allocating Built-In Arrays with new []

You also can use the new operator to dynamically allocate built-in arrays. The following statement dynamically allocates a 10-element integer array:

[Click here to view code image](#)

```
int* gradesArray{new int[10]{}};
```

This statement aims the int pointer gradesArray at the first element of the dynamically allocated array. The empty braced initializer following new int[10] **value initialize** the array's elements, which sets fundamental-type elements to 0, bools to false and pointers to nullptr. The braced initializer may also contain a comma-separated list of initializers for the array's elements. Value initializing an object calls its default constructor, if available. The rules become more complicated for objects that do not have default constructors. For more details, see the value-initialization rules at:

[Click here to view code image](#)

https://en.cppreference.com/w/cpp/language/value_.



The size of a built-in array created at compile time must be specified using an integral constant expression. However, a dynamically allocated array's size can be specified using *any* nonnegative integral expression that can be evaluated at execution time.

Releasing Dynamically Allocated Built-In Arrays with `delete[]`

To deallocate the memory to which `gradesArray` points, use the statement

```
delete[] gradesArray;
```

If the pointer points to a built-in array of objects, this statement first calls the destructor for each object in the array, then deallocates the memory for the entire array. As with `delete`, `delete[]` on a `nullptr` has no effect.

Err~~✗~~ If the preceding statement did *not* include the square brackets (`[]`) and `gradesArray` pointed to a built-in array of objects, the result is undefined. Some compilers call the destructor only for the first object in the array. **Using `delete` instead of `delete[]` for built-in arrays of objects results in undefined behavior. To ensure that every object in the array receives a destructor call, always use operator `delete[]` to delete memory allocated by `new[]`.** We'll show better techniques for managing dynamically allocated memory that enable you to avoid using `new` and `delete`.

Err~~✗~~ **If there is only one pointer to a block of dynamically allocated memory and the pointer goes out of scope or you assign it `nullptr` or a different memory address, a memory leak occurs. After deleting dynamically allocated memory, set the pointer's value to `nullptr` to indicate that it no longer points to memory in the free store. This ensures that your code cannot inadvertently access the previously allocated memory—doing so could cause subtle logic errors.**

Range-Based `for` Does Not Work with Dynamically Allocated Built-In Arrays

 You might be tempted to use C++11's range-based `for` statement to iterate over dynamically allocated arrays. Unfortunately, this will not compile. The compiler must know the number of elements at compile-time to iterate over an array with range-based `for`. As a workaround, you can create a C++20 `span` object that represents the dynamically allocated array and its number of elements, then iterate over the `span` object with range-based `for`.

11.5 Modern C++ Dynamic Memory Management —RAII and Smart Pointers

 A common design pattern is to allocate dynamic memory, assign the address of that memory to a pointer, use the pointer to manipulate the memory and deallocate the memory when it's no longer needed. If an exception occurs after successful memory allocation but before the `delete` or `delete[]` statement executes, a **memory leak** could occur.

 For this reason, the C++ Core Guidelines recommend that you manage resources like dynamic memory using **RAII—Resource Acquisition Is Initialization**.^{6,7} The concept is straightforward. For any resource that must be returned to the system when the program is done using it, the program should:

6. “R: Resource Management.” Accessed February 26, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource>.

7. “R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).” Accessed February 26, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-raii>.

- create the object as a local variable in a function—the object's constructor should allocate the resource,
- use that object as necessary in your program, then
- when the function call terminates, the object goes out of scope, which automatically calls the object's destructor to release the resource.

11.5.1 Smart Pointers

SE A C++11 **smart pointers** use RAII to manage dynamically allocated memory for you. The standard library header `<memory>` defines three smart pointer types:

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

A `unique_ptr` maintains a pointer to dynamically allocated memory. The class name contains “unique” because **a dynamically allocated object can belong to only one unique_ptr at a time**. When a `unique_ptr` object goes out of scope, its destructor uses `delete` (or `delete[]` for an array) to deallocate the memory that the `unique_ptr` manages. The rest of [Section 11.5](#) demonstrates `unique_ptr`, which we’ll also use in our [MyArray](#) case study. We introduce `shared_ptr` and `weak_ptr` in [Chapter 19, Other Topics and a Look Toward the Future of C++](#).

11.5.2 Demonstrating `unique_ptr`

[Figure 11.2](#) demonstrates a `unique_ptr` object that points to a dynamically allocated object of a custom class `Integer` (lines 7–22). For pedagogic purposes, the class’s constructor and destructor both display when they are called. Line 29 creates `unique_ptr` object `ptr` and initializes it with a pointer to a dynamically allocated `Integer` object that contains the value [14](#) 7. To initialize the `unique_ptr`, line 29 calls C++14’s **make_unique function template**, which allocates dynamic memory with operator `new`, then returns a `unique_ptr` to that memory.⁸ In this example, `make_unique<Integer>` returns a `unique_ptr<Integer>`—line 29 uses the `auto` keyword to infer `ptr`’s type from its initializer.

8. Prior to C++14, you’d pass the result of a `new` expression directly to `unique_ptr`’s constructor.

[Click here to view code image](#)

```
1 // fig11_02.cpp
2 // Demonstrating unique_ptr.
3 #include <iostream>
```

```
4 #include <memory>
5 using namespace std;
6
7 class Integer {
8 public:
9     // constructor
10    Integer(int i) : value{i} {
11        cout << "Constructor for Integer " <<
12    }
13
14    // destructor
15    ~Integer() {
16        cout << "Destructor for Integer " <<
17    }
18
19    int getValue() const {return value;} // .
20 private:
21    int value{0};
22 };
23
24 // use unique_ptr to manipulate Integer obj
25 int main() {
26     cout << "Creating a unique_ptr object th
27
28     // create a unique_ptr object and "aim"
29     auto ptr{make_unique<Integer>(7)};
30
31     // use unique_ptr to call an Integer mem
32     cout << "Integer value: " << ptr->getVal
33     << "\n\nMain ends\n";
34 }
```

< >

Creating a unique_ptr object that points to an I
Constructor for Integer 7
Integer value: 7

```
Main ends  
Destructor for Integer 7
```



Fig. 11.2 | Demonstrating unique_ptr.

Line 32 uses `unique_ptr`'s overloaded `->` operator to invoke function `getValue` on the `Integer` object that `ptr` manages. The expression `ptr->getValue()` also could have been written as:

```
(*ptr).getValue()
```

which uses `unique_ptr`'s overloaded `*` operator to dereference `ptr`, then uses the dot `(.)` operator to invoke function `getValue` on the `Integer` object.

Because `ptr` is a local variable in `main`, it's destroyed when `main` terminates. The `unique_ptr` destructor deletes the dynamically allocated `Integer` object, which calls the object's destructor. **The program releases the Integer object's memory, whether program control leaves the block normally via a return statement or reaching the end of the block—or as the result of an exception.**

SE Most importantly, using `unique_ptr` as shown here **prevents resource leaks**. For example, suppose a function returns a pointer aimed at some dynamically allocated object. Unfortunately, the function caller that receives this pointer might not delete the object, thus resulting in a **memory leak**. However, if the function returns a `unique_ptr` to the object, the object will be deleted automatically when the `unique_ptr` object's destructor gets called.

11.5.3 unique_ptr Ownership

SE Only one `unique_ptr` at a time can own a dynamically allocated object, so assigning one `unique_ptr` to another **transfers ownership** to the target `unique_ptr` on the assignment's left. The same is true when one `unique_ptr` is passed as an argument to another `unique_ptr`'s constructor. These operations use `unique_ptr`'s move assignment operator

and move constructor, which we discuss in [Section 11.6](#). **The last unique_ptr object that owns the dynamic memory will delete the memory.** This makes `unique_ptr` an ideal mechanism for returning ownership of dynamically allocated objects to client code. When the `unique_ptr` goes out of scope in the client code, the `unique_ptr`'s destructor deletes the dynamically allocated object. If that object has a destructor, it is called before the memory is returned to the system.

11.5.4 `unique_ptr` to a Built-In Array

You can also use a `unique_ptr` to manage a dynamically allocated built-in array, as we'll do in [Section 11.6](#)'s `MyArray` case study. For example, in the statement

[Click here to view code image](#)

```
auto ptr{make_unique<int []>(10)};
```

14 `make_unique`'s type argument is specified as `int[]`. So `make_unique` dynamically allocates a built-in array with the number of elements specified by its argument (10). By default, the `int` elements are value initialized to 0. The preceding statement uses `auto` to infer `ptr`'s type (`unique_ptr<int []>`), based on its initializer.

A `unique_ptr` that manages an array provides an **overloaded subscript operator ([])** to access its elements. For example, the statement

```
ptr[2] = 7;
```

assigns 7 to the `int` at `ptr[2]`, and the following statement displays that `int`:

```
cout << ptr[2] << "\n";
```

11.6 `MyArray` Case Study: Crafting a Valuable Class with Operator Overloading

Class development is an interesting, creative and intellectually challenging activity— always with the goal of crafting valuable classes. When we refer to “arrays” in this case study, we mean the built-in arrays we

discussed in [Chapter 7](#). These pointer-based arrays have many problems, including:

- C++ does not check whether an array index is out-of-bounds. A program can easily “walk off” either end of an array, likely causing a fatal runtime error if you forget to test for this possibility in your code.
 - Arrays of size n must use index values in the range 0 to $n - 1$. Alternate index ranges are not allowed.
 - You cannot input an entire array with the stream extraction operator (`>>`) or output one with the stream insertion operator (`<<`). You must read or write every element.⁹
9. In [Chapter 13](#), Standard Library Containers and Iterators, you’ll use C++ standard library functions to input and output entire containers of elements, such as `vectors` and `arrays`.
- Two arrays cannot be meaningfully compared with equality or relational operators. Array names are simply pointers to where the arrays begin in memory. Two arrays will always be at different memory locations.
 - When you pass an array to a general-purpose function that handles arrays of any size, you must pass the array’s size as an additional argument. As you saw in [Section 7.10](#), C++20’s `spans` help solve this problem.
 - **20** You cannot assign one array to another with the assignment operator(s).

With C++, you can implement more robust array capabilities via classes and operator overloading, as has been done with C++ standard library class templates `array` and `vector`. In this section, we’ll develop our own custom `MyArray` class that’s preferable to arrays. Internally, class `MyArray` will use a `unique_ptr` smart pointer to manage a dynamically allocated built-in array of `int` values.¹⁰

20 10. In this section, we’ll use operator overloading to craft a valuable class. In [Chapter 15](#), we’ll make it more valuable by converting it to a class template, and we’ll use C++20’s new concepts feature to add even more value.

We’ll create a powerful `MyArray` class with the following capabilities:

- `MyArrays` perform range checking when you access them via the subscript (`[]`) operator to ensure indices remain within their bounds.

Otherwise, the `MyArray` object will throw a standard library `out_of_bounds` exception.

- Entire `MyArrays` can be input or output with the overloaded stream extraction (`>>`) and stream insertion (`<<`) operators, without the client-code programmer having to write iteration statements.
- `MyArrays` may be compared to one another with the equality operators `==` and `!=`. The class could easily be enhanced to include the relational operators.
- `MyArrays` know their own size, making it easier to pass them to functions.
- `MyArray` objects may be assigned to one another with the assignment operator.
- `MyArrays` may be converted to `bool false` or `true` values to determine whether they are empty or contain elements.
- `MyArray` provides prefix and postfix increment (`++`) operators that add 1 to every element. We can easily add prefix and postfix decrement (`--`) operators.
- `MyArray` provides an addition assignment operator (`+=`) that adds a specified value to every element. The class could easily be enhanced to support the `-=`, `*=`, `/=` and `%=` assignment operators.

Class `MyArray` will demonstrate the **five special member functions** and the **unique_ptr smart pointer for managing dynamically allocated memory**. We'll use **RAII (Resource Acquisition Is Initialization)** throughout this example to manage the dynamically allocated memory resources. All dynamically allocated memory will be allocated by the class's constructors as `MyArray` objects are initialized and deallocated automatically by the class's destructor when `MyArray` objects go out of scope, **preventing memory leaks**. Our `MyArray` class is not meant to replace standard library class templates `array` and `vector`, nor is it meant to mimic their capabilities. It demonstrates key C++ language and library features that you'll find useful when you build your own classes.

11.6.1 Special Member Functions

Every class you define can have five **special member functions**, each of which we define in class MyArray:

- a **copy constructor**,
- a **copy assignment operator**,
- a **move constructor**,
- a **move assignment operator**, and
- a **destructor**.

The copy constructor and copy assignment operator implement the class's **copy semantics**—that is, how to copy a MyArray when it is passed by value to a function, returned by value from a function or assigned to another MyArray. The move constructor and move assignment operator implement the class's **move semantics**, which eliminate costly unnecessary copies of objects that are about to be destroyed. We discuss the details of these special member functions as we encounter the need for them throughout this case study.

11.6.2 Using Class MyArray

The program of Figs. 11.3–11.5 demonstrates class MyArray and its rich selection of overloaded operators. The code in Fig. 11.3 tests the various MyArray capabilities. We present the class definition in Fig. 11.4 and each of its member-function definitions in Fig. 11.5. We've broken the code and outputs into small segments for discussion purposes. For pedagogic purposes, many of class MyArray's member functions, including all of special member functions, display output to show when they're called.

Function `getArrayByValue`

Later in this program, we'll call the `getArrayByValue` function (Fig. 11.3, lines 10–13) to create a local MyArray object by calling MyArray's constructor that receives an **initializer list**. Function `getArrayByValue` returns that local object *by value*.

[Click here to view code image](#)

```
1 // fig11_03.cpp
```

```
2 // MyArray class test program.
3 #include <iostream>
4 #include <stdexcept>
5 #include <utility> // for std::move
6 #include "MyArray.h"
7 using namespace std;
8
9 // function to return a MyArray by value
10 MyArray getArrayByValue() {
11     MyArray localInts{10, 20, 30}; // create
12     return localInts; // return by value cre-
13 }
14
```



Fig. 11.3 | MyArray class test program.

Creating MyArray Objects and Displaying Their Size and Contents

Lines 16–17 create objects ints1 with seven elements and ints2 with 10 elements. Each calls the MyArray constructor that receives the number of elements and initializes the elements to zeros. Lines 20–21 display ints1's size then output its contents using MyArray's **overloaded stream insertion operator (<<)**. Lines 24–25 do the same for ints2. Each statement uses << to output two string literals, a size_t and a MyArray object.

[Click here to view code image](#)

```
15 int main() {
16     MyArray ints1(7); // 7-element MyArray;
17     MyArray ints2(10); // 10-element MyArray
18
19     // print ints1 size and contents
20     cout << "\nints1 size: " << ints1.size()
21         << "\ncontents: " << ints1; // uses o-
22
```

```
23     // print ints2 size and contents
24     cout << "\n\nints2 size: " << ints2.size
25         << "\ncontents: " << ints2; // uses o·
26
```



```
MyArray(size_t) constructor
MyArray(size_t) constructor

ints1 size: 7
contents: {0, 0, 0, 0, 0, 0, 0}

ints2 size: 10
contents: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Using Parentheses Rather Than Braces to Call the Constructor

So far, we've used braced initializers, {}, to pass arguments to constructors. Lines 16–17 use parentheses, (), to call the `MyArray` constructor that receives a `size`. We do this because our class—like the standard library's `array` and `vector` classes—also supports constructing a `MyArray` from a braced initializer list containing the `MyArray`'s element values. When the compiler sees a statement like:

```
MyArray ints1{7};
```

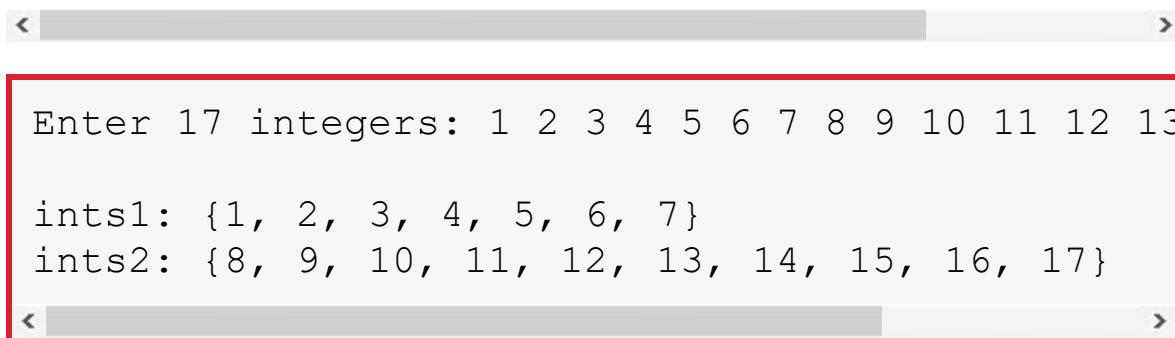
it invokes the constructor that accepts the braced-initializer list of integers, not the single-argument constructor that receives the `size`.

Using the Overloaded Stream Extraction Operator to Fill a `MyArray`

Next, line 28 prompts the user to enter 17 integers. Line 29 uses the `MyArray` **overloaded stream extraction operator (>>)** to read the first seven values into `ints1` and the remaining 10 values into `ints2` (recall that each `MyArray` knows its own `size`). Line 31 displays each `MyArray`'s updated contents using the **overloaded stream insertion operator (<<)**.

[Click here to view code image](#)

```
27 // input and print ints1 and ints2
28 cout << "\n\nEnter 17 integers: ";
29 cin >> ints1 >> ints2; // uses overloaded >
30
31 cout << "\nints1: " << ints1 << "nints2: "
32
```



The screenshot shows a terminal window with a red border. Inside, the program prompts the user to enter 17 integers. The user enters the numbers 1 through 17 separated by spaces. The program then outputs two MyArray objects: ints1 containing {1, 2, 3, 4, 5, 6, 7} and ints2 containing {8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}. The terminal has scroll bars at the bottom.

```
Enter 17 integers: 1 2 3 4 5 6 7 8 9 10 11 12 13
ints1: {1, 2, 3, 4, 5, 6, 7}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Using the Overloaded Inequality Operator (`!=`)

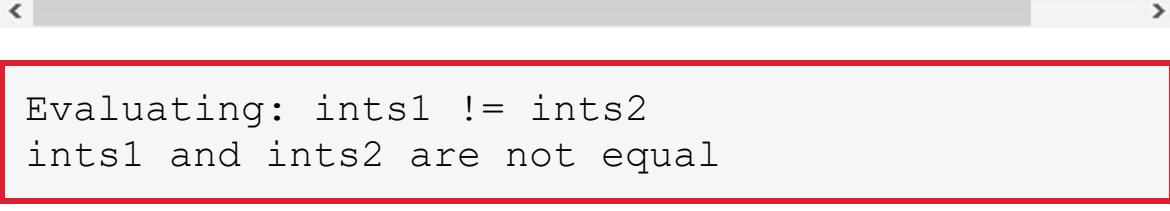
Line 36 tests `MyArray`'s **overloaded inequality operator (`!=`)** by evaluating the condition

```
ints1 != ints2
```

20 The program output shows that the `MyArray` objects are not equal. Two `MyArray` objects will be equal if they have the same number of elements and their corresponding element values are identical. As you'll see, we define only `MyArray`'s overloaded `==` operator. In C++20, the compiler autogenerated `!=` if you provide an `==` operator for your type—`!=` simply returns the opposite of `==`.

[Click here to view code image](#)

```
33 // use overloaded inequality (!=) operator
34 cout << "\n\nEvaluating: ints1 != ints2\n";
35
36 if (ints1 != ints2) {
37     cout << "ints1 and ints2 are not equal\n"
38 }
```



```
Evaluating: ints1 != ints2
ints1 and ints2 are not equal
```

Initializing a New `MyArray` with a Copy of an Existing `MyArray`

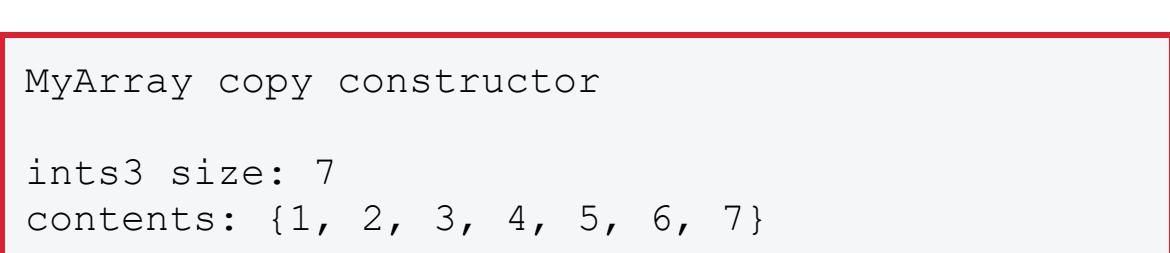
Line 41 instantiates the `MyArray` object `ints3` and initializes it with a copy of `ints1`'s data. This invokes the `MyArray` **copy constructor** to copy `ints1`'s elements into `ints3`. A **copy constructor** is invoked whenever a copy of an object is needed, such as:

- passing an object by value to a function,
- returning an object by value from a function, or
- initializing an object with a copy of another object of the same class.

Line 44 displays `ints3`'s size and contents to confirm that `ints3`'s elements were set correctly by the **copy constructor**.

[Click here to view code image](#)

```
40 // create MyArray ints3 by copying ints1
41 MyArray ints3{ints1}; // invokes copy const.
42
43 // print ints3 size and contents
44 cout << "\nints3 size: " << ints3.size() <<
45
```



```
MyArray copy constructor

ints3 size: 7
contents: {1, 2, 3, 4, 5, 6, 7}
```

SE A 11 When a class such as `MyArray` contains both a **copy constructor** and a **move constructor**, the compiler chooses the correct one to use based on the context. In line 41, the compiler chose `MyArray`'s **copy constructor** because variable names, like `ints1`, are *lvalues*. As you'll soon see, a **move constructor** receives an **rvalue reference**, which is part of C++11's **move semantics**. **An rvalue reference may not refer to an lvalue.**

The **copy constructor** can also be invoked by writing line 41 as:

```
MyArray ints3 = ints1;
```

In an object's definition, the equal sign does *not* indicate assignment. It invokes the single-argument copy constructor, passing as the argument the value to the = symbol's right.

Using the Overloaded Copy Assignment Operator (=)

SE A Line 48 assigns `ints2` to `ints1` to test the **overloaded copy assignment operator (=)**. Built-in arrays cannot handle this assignment. An array's name is not a modifiable *lvalue*, so you cannot assign one array's name to another—such an assignment causes a compilation error. Line 50 displays both objects' contents to confirm that they're now identical. `MyArray` `ints1` initially held seven integers, but the overloaded operator **resizes** the dynamically allocated built-in array to hold a copy of `ints2`'s 10 elements. As with **copy constructors** and **move constructors**, if a class **contains both a copy assignment operator and a move assignment operator, the compiler chooses which one to call based on the arguments**. In this case, `ints2` is a variable and thus an *lvalue*, so the copy assignment operator is called. Note in the output that line 48 also resulted in calls to the `MyArray` copy constructor and destructor—you'll see why when we present the assignment operator's implementation in [Section 11.6.6](#).

[Click here to view code image](#)

```
46 // use overloaded copy assignment (=) opera
47 cout << "\n\nAssigning ints2 to ints1:\n";
48 ints1 = ints2; // note target MyArray is sm
49
```

```
50     cout << "\nints1: " << ints1 << "\nints2: "
51
```

```
<          >
Assigning ints2 to ints1:
MyArray copy assignment operator
MyArray copy constructor
MyArray destructor

ints1: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Using the Overloaded Equality Operator (==)

Line 55 compares ints1 and ints2 with the **overloaded equality operator (==)** to confirm they are indeed identical after the assignment in line 48.

[Click here to view code image](#)

```
52 // use overloaded equality (==) operator
53 cout << "\n\nEvaluating: ints1 == ints2\n";
54
55 if (ints1 == ints2) {
56     cout << "ints1 and ints2 are equal\n\n";
57 }
58
```

```
<          >
Evaluating: ints1 == ints2
ints1 and ints2 are equal
```

Using the Overloaded Subscript Operator ([])

Line 60 uses the **overloaded subscript operator ([])** to refer to ints1[5] —an *in-range* element of ints1. This indexed (subscripted) name is used to

get the value stored in `ints1[5]`. Line 64 uses `ints1[5]` on an assignment's left side as a modifiable *lvalue*¹¹ to assign a new value, 1000, to element 5 of `ints1`. We'll see that `operator[]` returns a reference to use as the modifiable *lvalue* after confirming 5 is a valid index. Line 70 attempts to assign 1000 to `ints1[15]`. **This index is outside `int1`'s bounds, so the overloaded `operator[]` throws an `out_of_range` exception.** Lines 72–74 catch the exception and display its error message by calling the exception's `what` member function.

¹¹ Recall that an *lvalue* can be declared `const`, in which case it would not be modifiable.

[Click here to view code image](#)

```
59 // use overloaded subscript operator to cre.
60 cout << "ints1[5] is " << ints1[5];
61
62 // use overloaded subscript operator to cre.
63 cout << "\n\nAssigning 1000 to ints1[5]\n";
64 ints1[5] = 1000;
65 cout << "ints1: " << ints1;
66
67 // attempt to use out-of-range subscript
68 try {
69     cout << "\n\nAttempt to assign 1000 to i:
70     ints1[15] = 1000; // ERROR: subscript ou:
71 }
72 catch (const out_of_range& ex) {
73     cout << "An exception occurred: " << ex.
74 }
75
```

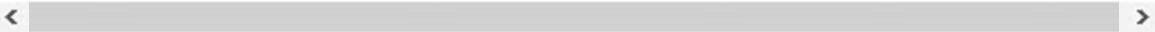


```
ints1[5] is 13
```

```
Assigning 1000 to ints1[5]
```

```
ints1: {8, 9, 10, 11, 12, 1000, 14, 15, 16, 17}
```

```
Attempt to assign 1000 to ints1[15]
An exception occurred: Index out of range
```



The **array subscript operator []** is not restricted for use only with arrays. It also can be used, for example, to select elements from other kinds of *container classes* that maintain collections of items, such as strings (collections of characters) and maps (collections of key-value pairs, which we'll discuss in [Chapter 13](#), Standard Library Containers and Iterators). Also, when **overloaded operator[]** functions are defined, *indices are not required to be integers*. In [Chapter 13](#), we discuss the standard library map class that allows indices of other types, such as strings.

[Creating MyArray ints4 and Initializing It With the MyArray Returned By Function getArrayByValue](#)

Line 79 initializes MyArray ints4 with the result of calling function `getArrayByValue` (lines 10–13), which creates a local MyArray containing 10, 20 and 30, then returns it *by value*. Then, line 81 displays the new MyArray's size and contents.

[Click here to view code image](#)

```
76 // initialize ints4 with contents of the My
77 // getArrayByValue; print size and contents
78 cout << "\nInitialize ints4 with temporary "
79 MyArray ints4{getArrayByValue()};
80
81 cout << "\nints4 size: " << ints4.size() <<
82
```



```
Initialize ints4 with temporary MyArray object
MyArray(initializer_list) constructor
```

```
ints4 size: 3  
contents: {10, 20, 30}
```

Named Return Value Optimization (NRVO)

Recall from function `getArrayByValue`'s definition (lines 10–13) that it creates and initializes a local `MyArray` using the constructor that receives an `initializer_list` of `int` values (line 11). This constructor displays

[Click here to view code image](#)

```
MyArray(initializer_list) constructor
```

each time it's called. Next, `getArrayByValue` returns that local array *by value* (line 12). You might expect that returning an object by value would make a temporary copy of the object for use in the caller. If it did this would call `MyArray`'s copy constructor to copy the local `MyArray`. You also might expect the local `MyArray` object to go out of scope and have its destructor called as `getArrayByValue` returns to its caller. However, neither the copy constructor nor the destructor displayed lines of output here.

Perf  17 This is due to a compiler performance optimization called **named return value optimization (NRVO)**. When the compiler sees that a local object is constructed, returned from a function by value then used to initialize an object in the caller, the compiler instead constructs the object directly in the caller where it will be used, eliminating the temporary object and extra constructor and destructor calls mentioned above. Prior to C++17 this was an optional optimization, but this optimization is required as of C++17.^{12,13}

12. Sy Brand. “Guaranteed Copy Elision Does Not Elide Copies.” C++ Team Blog, February 18, 2019. Accessed February 24, 2021. <https://devblogs.microsoft.com/cppblog/guaranteed-copy-elision-does-not-elide-copies/>.

13. Richard Smith, “Guaranteed copy elision through simplified value categories,” September 27, 2015. Accessed February 24, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html>.

Creating `MyArray ints5` and Initializing It With the *rvalue* Returned By Function `std::move`

A **copy constructor** is called when you initialize one `MyArray` with another that's represented by an *lvalue*. A copy constructor copies its arguments contents. This is similar to a copy-and-paste operation in a text editor—after the operation, you have two copies of the data.

Perf C++ also supports **move semantics**,¹⁴ which help the compiler eliminate the overhead of unnecessarily copying objects. A move is similar to a cut-and-paste operation in a text editor—the data gets *moved* from the cut location to the paste location. A **move constructor** moves into a new object the resources of an object that's no longer needed. Such a constructor receives a C++11 **rvalue reference**, which you'll see is declared with `TypeName&&`. *Rvalue* references may refer only to *rvalues*. Typically, these are temporary objects or objects that are about to be destroyed—called **expiring values** or **xvalues**.

14. Klaus Iglberger, “Back to Basics: Move Semantics,” YouTube Video, June 16, 2019, <https://www.youtube.com/watch?v=St0MNEU5b0o>.

11 Line 86 uses class `MyArray`'s **move constructor** to initialize `MyArray ints5`, then line 88 displays the size and contents of the new `MyArray`. The object `ints4` is an *lvalue*—so it cannot be passed directly to `MyArray`'s move constructor. If you no longer need an object's resources, you can **convert it from an lvalue to an rvalue reference** by passing the object to the C++11 standard library function `std::move` (from header `<utility>`). **This function casts its argument to an rvalue reference**,¹⁵ telling the compiler that `ints4`'s contents are no longer needed. So, line 86 forces `MyArray`'s **move constructor** to be called and `ints4`'s contents are *moved* into `ints5`.

15. More specifically, this is called *xvalue* (for expiring value).

[Click here to view code image](#)

```
83 // convert ints4 to an rvalue reference wit:
84 // use the result to initialize MyArray int
85 cout << "\n\nInitialize ints5 with the resu
86 MyArray ints5{std::move(ints4)}; // invokes
87
```

```
88     cout << "\nints5 size: " << ints5.size() <<
89     cout << "\n\nSize of ints4 is now: " << int
90
```

Initialize ints5 with the result of std::move(in
MyArray move constructor

```
ints5 size: 3
contents: {10, 20, 30}
```

```
Size of ints4 is now: 0
```

It's recommended that you use std::move as shown here **only if you know the source object passed to std::move will never be used again**. Once an object has been moved, two valid operations can be performed with it:

- destroying it, and
- using it on the left side of an assignment to give it a new value.

In general, you should not call member functions on a moved-from object. We do so in line 89 only to prove that the **move constructor** indeed moved ints4's resources—the output shows that ints4's size is now 0.

Assigning MyArray ints5 to ints4 with the Move Assignment Operator

Lines 93 and 95 use class MyArray's **move assignment operator** to move ints5's contents (10, 20 and 30) back into ints4 then display the size and contents of ints4. Line 93 *explicitly converts the lvalue ints5 to an rvalue reference* using std::move. This indicates that ints5 no longer needs its resources, so the compiler can move them into ints4. In this case, the compiler calls MyArray's **move assignment operator**. For demo purposes only, we output ints5's size to show that the **move assignment operator** indeed moved its resources. Again, **you should not call member functions on a moved-from object**.

[Click here to view code image](#)

```
91 // move contents of ints5 into ints4
92 cout << "\n\nMove ints5 into ints4 via move
93 ints4 = std::move(ints5); // invokes move a
94
95 cout << "\nints4 size: " << ints4.size() <<
96 cout << "\n\nSize of ints5 is now: " << int
97
```

```
< >
Move ints5 into ints4 via move assignment
MyArray move assignment operator
```

```
ints4 size: 3
contents: {10, 20, 30}
```

```
Size of ints5 is now: 0
```

Converting MyArray ints5 to a bool to Test Whether It's Empty

Many programming languages allow you to use a container-class object like a `MyArray` as a condition to determine whether the container has elements. For class `MyArray`, we defined a `bool` conversion operator that returns `true` if the `MyArray` object contains elements (i.e., its `size` is greater than 0) and `false` otherwise. In contexts that require `bool` values, such as control statement conditions, C++ can invoke an object's `bool` conversion operator implicitly. This is known as a **contextual conversion**. Line 99 uses the `MyArray` `ints5` as a condition, which calls `MyArray`'s `bool` conversion operator. Since we just moved `ints5`'s resources into `ints4`, `ints5` is now empty, and the operator returns `false`. *Once again, you should not call member functions on moved-from objects*—we do so here only to prove that `ints5`'s resources have been moved.

[Click here to view code image](#)

```
98 // check if ints5 is empty by contextually
99 if (ints5) {
100     cout << "\n\nints5 contains elements\n"
101 }
102 else {
103     cout << "\n\nints5 is empty\n";
104 }
105
```



```
ints5 is empty
```

Preincrementing Every `ints4` Element with the Overloaded `++` Operator

Some libraries support “**broadcast**” operations that apply the same operation to every element of a data structure. For example, consider the popular high-performance Python programming language library **NumPy**. This library’s **ndarray (n-dimensional array) data structure** overloads many arithmetic operators. They conveniently perform mathematical operations on *every* element of an ndarray. In NumPy, the following Python code adds one to every element of the ndarray named `numbers` (no iteration statement is required):

[Click here to view code image](#)

```
numbers += 1 # Python does not have a ++ operator
```

To demonstrate preincrement and postincrement, we’ve added a similar capability to class `MyArray`. Line 107 displays `ints4`’s current contents, then line 108 uses the expression `++ints4` to preincrement the entire `MyArray`, by adding one to every element. This expression’s result is the updated `MyArray`. We then use `MyArray`’s **overloaded stream insertion operator (`<<`)** to display the contents.

[Click here to view code image](#)

```
106 // add one to every element of ints4 using
107 cout << "\nints4: " << ints4;
108 cout << "\npreincrementing ints4: " << ++ints4;
109
```

```
<                                >
ints4: {10, 20, 30}
preincrementing ints4: {11, 21, 31}
```

Postincrementing Every `ints4` Element with the Overloaded `++` Operator

Line 111 postincrements the entire `MyArray` with the expression `ints4++`. Recall that a postincrement returns its operand's **previous value**, as confirmed by the program's output. The outputs showing that `MyArray`'s **copy constructor** and **destructor** were called are from the postincrement operator's implementation, which we'll discuss in [Section 11.6.14](#).

[Click here to view code image](#)

```
110 // add one to every element of ints4 using
111 cout << "\n\npostincrementing ints4: " <<
112 cout << "\nints4 now contains: " << ints4;
113
```

```
<                                >
postincrementing ints4: MyArray copy constructor
{11, 21, 31}
MyArray destructor

ints4 now contains: {12, 22, 32}
```

Adding a Value to Every `ints4` Element with the Overloaded `+=` Operator

Class `MyArray` also provides a broadcast-based **overloaded addition assignment operator (`+=`)** for adding an `int` value to every `MyArray` element. Line 115 adds 7 to every `ints4` element then displays its new contents. Note that the non-overloaded versions of `++` and `+=` still work on individual `MyArray` elements too—those are simply `int` values.

[Click here to view code image](#)

```
114      // add a value to every element of ints
115      cout << "\n\nAdd 7 to every ints4 elemen
116  }
```

< >

```
Add 7 to every ints4 element: {19, 29, 39}
```

Destroying the `MyArray` Objects That Remain

When function `main` terminates, the **destructors** are called for the five `MyArray` objects created in `main`, producing the last five lines of the program's output.

```
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
```

11.6.3 `MyArray` Class Definition

Now, let's walk through `MyArray`'s header (Fig. 11.4). As we refer to each member function in the header, we discuss that function's implementation in Fig. 11.5. We've broken the member-function implementation file into small segments for discussion purposes.

In Fig. 11.4, lines 53–54 declare `MyArray`'s private data members:

- `m_size` stores its number of elements, and

- `m_ptr` is a `unique_ptr` that manages a dynamically allocated pointer-based `int` array containing the `MyArray` object's elements. When a `MyArray` goes out of scope, its destructor will call the `unique_ptr`'s destructor which will automatically delete the dynamically allocated memory.

Throughout this class's member-function implementations, we use some of C++'s declarative, functional-style programming capabilities discussed in [Chapters 6](#) and [7](#). We also introduce three additional standard library algorithms—`copy`, `for_each` and `equal`.

[Click here to view code image](#)

```

1 // Fig. 11.4: MyArray.h
2 // MyArray class definition with overloaded
3 #pragma once
4 #include <initializer_list>
5 #include <iostream>
6 #include <memory>
7
8 class MyArray final {
9     // overloaded stream extraction operator
10    friend std::istream& operator>>(std::ist
11
12    // used by copy assignment operator to i
13    friend void swap(MyArray& a, MyArray& b)
14
15 public:
16    explicit MyArray(size_t size); // constr
17
18    // construct a MyArray with a braced-init
19    explicit MyArray(std::initializer_list<i
20
21    MyArray(const MyArray& original); // cop
22    MyArray& operator=(const MyArray& right)
23
24    MyArray(MyArray&& original) noexcept; //

```

```

25     MyArray& operator=(MyArray&& right) noexcept
26
27     ~MyArray(); // destructor
28
29     size_t size() const noexcept {return m_size;}
30     std::string toString() const; // create
31
32     // equality operator
33     bool operator==(const MyArray& right) const;
34
35     // subscript operator for non-const objects
36     int& operator[](size_t index);
37
38     // subscript operator for const objects
39     const int& operator[](size_t index) const;
40
41     // convert MyArray to a bool value: true
42     explicit operator bool() const noexcept;
43
44     // preincrement every element, then return
45     MyArray& operator++();
46
47     // postincrement every element, and return
48     MyArray operator++(int);
49
50     // add value to every element, then return
51     MyArray& operator+=(int value);
52 private:
53     size_t m_size{0}; // pointer-based array
54     std::unique_ptr<int[]> m_ptr; // smart pointer
55 };
56
57 // overloaded operator<< is not a friend--defined
58 std::ostream& operator<<(std::ostream& out,

```



Fig. 11.4 | MyArray class definition with overloaded operators. (Part 2)

of 2.)

11.6.4 Constructor That Specifies a MyArray's Size

Line 16 of Fig. 11.4

[Click here to view code image](#)

```
explicit MyArray(size_t size); // construct a MyA:
```



declares a **constructor** that specifies the number of MyArray elements. The constructor's definition (Fig. 11.5, lines 16–19) performs several tasks:

- Line 17 initializes the `m_size` member using the argument `size`.
- Line 17 initializes the `m_ptr` member to a `unique_ptr` returned by the standard library's `make_unique` function template (see [Section 11.5](#)). Here, we use it to create a dynamically allocated `int` array of `size` elements. The function `make_unique` **value initializes the dynamic memory** it allocates, so the `int` array's elements are set to 0.
- For pedagogic purposes, line 18 displays that the constructor was called. We do this in all of MyArray's **special member functions and other constructors** to give you visual confirmation that the functions are being called.

[Click here to view code image](#)

```
1 // Fig. 11.5: MyArray.cpp
2 // MyArray class member- and friend-function definitions
3 #include <algorithm>
4 #include <initializer_list>
5 #include <iostream>
6 #include <memory>
7 #include <span>
8 #include <sstream>
9 #include <stdexcept>
10 #include <utility>
11 #include "fmt/format.h" // In C++20, this will be
```

```
12 #include "MyArray.h" // MyArray class defin
13 using namespace std;
14
15 // MyArray constructor to create a MyArray
16 MyArray::MyArray(size_t size)
17     : m_size{size}, m_ptr{make_unique<int>(new int[m_size])}
18     cout << "MyArray(size_t) constructor"
19 }
20
```



Fig. 11.5 | MyArray class member- and friend-function definitions.

11 11.6.5 C++11 Passing a Braced Initializer to a Constructor

In Fig. 6.2, we initialized a `std::array` object with braced-initializer list, as in

[Click here to view code image](#)

```
array<int, 5> n{32, 27, 64, 18, 95};
```

You can use **braced initializers** for objects of your own classes by providing a **constructor** with a `std::initializer_list` parameter, as declared in line 19 of Fig. 11.4:

[Click here to view code image](#)

```
explicit MyArray(std::initializer_list<int> list)
```



The `std::initializer_list` class template is defined in the header `<initializer_list>`. With this constructor, we can create `MyArray` objects that initialize their elements as they're constructed, as in:

```
MyArray ints{10, 20, 30};
```

or

```
MyArray ints = {10, 20, 30};
```

SE A Each creates a three-element `MyArray` containing 10, 20 and 30. In a class that provides an `initializer_list` constructor, the class's other single-argument constructors must be called using parentheses rather than braces. The braced initialization constructor (lines 22–29) has one `initializer_list<int>` parameter named `list`. You can determine `list`'s number of elements by calling its `size` member function (line 23).

[Click here to view code image](#)

```
21 // MyArray constructor that accepts an init
22 MyArray::MyArray(initializer_list<int> list
23   : m_size{list.size()}, m_ptr{make_unique<
24     cout << "MyArray(initializer_list) const"
25
26   // copy list argument's elements into m_
27   // m_ptr.get() returns the int array's s
28   copy(begin(list), end(list), m_ptr.get())
29 }
30
```



To copy each initializer list value into the new `MyArray` object, line 28 uses the standard library `copy` algorithm (from header `<algorithm>`) to copy each `initializer_list` element into the new `MyArray`. The algorithm copies each element in the range specified by its first two arguments—the beginning and end of the `initializer_list`. These elements are copied into the destination specified by `copy`'s third argument. The `unique_ptr`'s `get member function` returns the `int*` that points to the first element of the `MyArray`'s underlying `int` array.

11.6.6 Copy Constructor and Copy Assignment Operator

CG Sections 9.15 and 9.17 introduced the **compiler-generated default copy assignment operator** and **default copy constructor**. These performed **memberwise copy operations by default**. The C++ Core Guidelines

recommend designing your classes such that the compiler can autogenerate the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. This is known as the **Rule of Zero**.¹⁶ You can accomplish this by composing each class's data using fundamental-type members and objects of classes that do not require you to implement custom resource processing, such as standard library classes like `array` and `vector`, which each use RAII to manage resources.

16. "C.20: If you can avoid defining any default operations, do." Accessed March 3, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero>.

The Rule of Five

 The default special member functions work well for fundamental-type values like `ints` and `doubles`. But what about objects of types that manage their own resources, such as pointers to dynamically allocated memory? Classes that manage their own resources should explicitly define the five special member functions. The C++ Core Guidelines state that if a class requires one special member function, it should define them all,¹⁷ as we do in this case study. This is known as the **Rule of Five**.

17. "C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all." Accessed March 3, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-five>.

11 Even for classes with the compiler-generated special member functions, some experts recommend declaring them explicitly in the class definition with `= default` (introduced in Chapter 10). This is called the **Rule of Five defaults**.¹⁸ You also can explicitly remove compiler-generated special member functions to prevent the specified functionality by following their function prototypes with C++11's `= delete`. Class `unique_ptr` actually does this for the copy constructor and copy assignment operator.

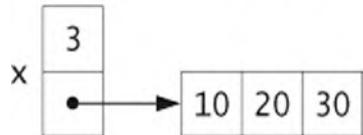
18. Scott Meyers, "A Concern about the Rule of Zero," March 13, 2014. Accessed March 6, 2021.
<http://scottmeyers.blogspot.com/2014/03/a-concern-about-rule-of-zero.html>.

Shallow Copy

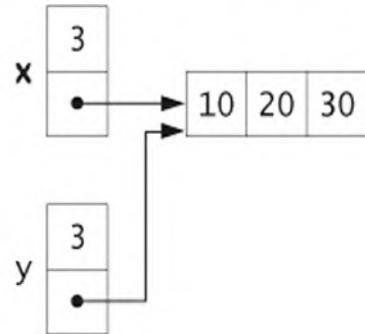
The compiler-generated copy constructor and copy assignment operator

perform memberwise **shallow copies**. If the member is a pointer to dynamically allocated memory, only the address in the pointer is copied. In the following diagram, consider the object `x`, which contains its number of elements (3) and a pointer to a dynamically allocated array. For this discussion, let's assume the pointer member is just an `int*`, not a `unique_ptr`.

Before `x` is shallow copied into `y`



After `x` is shallow copied into `y`

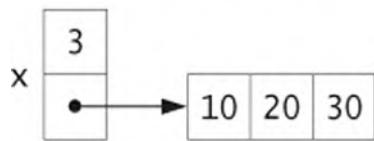


Err~~✗~~ Now, assume we'd like to copy `x` into a new object `y`. If the **copy constructor** simply copied the pointer in `x` into the target object `y`'s pointer, both would point to the same **dynamically allocated memory**, as in the right side of the diagram. **The first destructor to execute would delete the memory that is shared between these two objects. The other object's pointer would then point to memory that's no longer allocated.** This situation is called a **dangling pointer**, and typically would result in a serious runtime error (such as early program termination) if the program were to dereference that pointer—accessing deleted memory is undefined behavior.

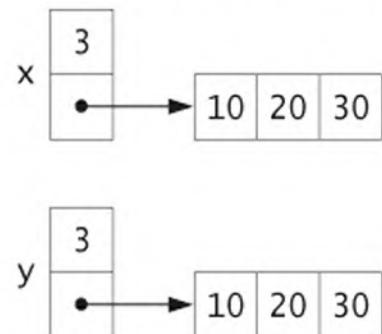
Deep Copy

Err~~✗~~ In classes that manage their own resources, copying must be done carefully to avoid the pitfalls of shallow copy. Classes that manage their objects' resources should define their own **copy constructor** and **overloaded copy assignment operator** to perform **deep copies**. The following diagram shows that after the object `x` is deep copied into the object `y`, both objects have their own copies of the dynamically allocated array containing `10`, `20` and `30`. Not providing a copy constructor and overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a potential logic error.

Before `x` is deep copied into `y`



After `x` is deep copied into `y`



Implementing the Copy Constructor

Line 21 of Fig. 11.4

[Click here to view code image](#)

```
MyArray(const MyArray& original); // copy constructor
```

◀ ▶

SE  declares the class's **copy constructor** (defined lines 32–39). The copy constructor's argument must be a **const reference** to prevent the constructor from modifying the argument object's data.

[Click here to view code image](#)

```
31  // copy constructor: must receive a reference
32  MyArray::MyArray(const MyArray& original)
33      : m_size{original.size()}, m_ptr{make_unique<int>(new int[m_size])}
34      cout << "MyArray copy constructor\n";
35
36  // copy original's elements into m_ptr's
37  const span<const int> source{original.m_ptr};
38  copy(begin(source), end(source), m_ptr.get());
39 }
40
```

◀ ▶

When the **copy constructor** is called to initialize a new `MyArray` by copying an existing one, it performs the following tasks:

- Line 33 initializes the `m_size` member using the return value of `original.size` member function.
- Line 33 initializes the `m_ptr` member to a `unique_ptr` returned by the standard library's `make_unique` function template, which creates a dynamically allocated `int` array containing `original.size()` elements.
- Line 34 outputs that the **copy constructor** was called.
- Recall that a **span** is a view into a contiguous collection of items, such as an array. Line 37 creates a **span** named `source` representing the argument `MyArray`'s **dynamically allocated int array** from which we'll copy elements.
- Line 38 uses the **copy algorithm** to copy the elements in the range represented by the beginning and end of the `source` span into the `MyArray`'s underlying `int` array.

[Copy Assignment Operator \(=\)](#)

Line 22 of [Fig. 11.4](#)

[Click here to view code image](#)

```
MyArray& operator=(const MyArray& right); // copy
```

declares the class's **overloaded copy assignment operator (=)**.¹⁹ This function's definition (lines 42–47) enables one `MyArray` to be assigned to another, copying the contents from the right operand into the left. When the compiler sees the statement

19. This copy assignment operator ensures that the `MyArray` object is not modified and no memory resources are leaked if an exception occurs. This is known as a strong exception guarantee, which we discuss in [Section 12.3](#).

```
ints1 = ints2;
```

it invokes member function `operator=` with the call

```
ints1.operator=(ints2)
```

[Click here to view code image](#)

```
41 // copy assignment operator: implemented with
42 MyArray& MyArray::operator=(const MyArray&
43     cout << "MyArray copy assignment operator"
44     MyArray temp{right}; // invoke copy constructor
45     swap(*this, temp); // exchange contents
46     return *this;
47 }
48
```



We could implement the **overloaded copy assignment operator** similarly to the **copy constructor**. However, there's an elegant way to use the **copy constructor** to implement the **overloaded copy assignment operator**—the **copy-and-swap idiom**.^{20,21} The idiom operates as follows:

20. Sutter, Herb. “Exception-Safe Class Design, Part 1: Copy Assignment.” Accessed February 25, 2021. <http://www.gotw.ca/gotw/059.htm>.
21. “What is the copy-and-swap idiom?” Accessed February 25, 2021. <https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom>.

- First, it copies the argument `right` into a local `MyArray` object (`temp`) using the **copy constructor** (line 44). If this operation fails to allocate memory for `temp`'s array, a **bad_alloc exception** will occur. In this case, the **overloaded copy assignment operator** will immediately terminate without modifying the object on the assignment's left.
- Line 45 uses class `MyArray`'s `friend` function `swap` (defined in lines 165–168) to exchange the contents of `*this` (the object on the assignment's left) with `temp`.
- Finally, the function returns a reference to the current object (`*this` in line 46), enabling cascaded `MyArray` assignments such as `x=y= z`.

When the function returns to its caller, the `temp` object's **destructor** is called to **release the memory** managed by the `temp` object's `unique_ptr`. Line 44's copy constructor call and the destructor call when `temp` goes out of scope are the reason for the two additional lines of output you saw when we demonstrated assigning `ints2` to `ints1` in line 48 of Fig. 11.3.

11.6.7 Move Constructor and Move Assignment Operator

Perf  Often an object being copied is about to be destroyed, such as a local object returned from a function by value. It's more efficient to move that object's contents into the destination object to eliminate the copying overhead. That's the purpose of the **move constructor** and **move assignment operator** declared in lines 24–25 of Fig. 11.4

[Click here to view code image](#)

```
MyArray(MyArray&& original) noexcept; // move constructor
MyArray& operator=(MyArray&& right) noexcept; // move assignment operator
```



Each receives an **rvalue reference** declared with `&&` to distinguish it from an **lvalue reference** `&`. **Rvalue references** help implement **move semantics**. Rather than copying the argument, the **move constructor** and **move assignment operator** each move their argument object's data, leaving the original object in a state that can be destructed properly.

noexcept Specifier

Err  As of C++11, if a function does not throw any exceptions *and* does not call any functions that throw exceptions, you should explicitly state that the function does not throw exceptions.²² Simply add **noexcept** after the function's signature in both the prototype and the definition. For a `const` member function, keyword `noexcept` must be placed after `const`. If a `noexcept` function calls another function that throws an exception and the `noexcept` function does not handle that exception, the program terminates immediately.

²². “F.6: If your function may not throw, declare it `noexcept`.” Accessed February 25, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-noexcept>.

The `noexcept` specification can optionally be followed by parentheses containing a `bool` expression that evaluates to `true` or `false`. `noexcept` by itself is equivalent to `noexcept(true)`. Following a function's signature with `noexcept(false)` indicates that the class's designer has thought about whether the function might throw exceptions and has decided it

might. In such cases, client code programmers can decide whether to wrap calls to the function in `try` statements.

Class `MyArray`'s Move Constructor

The **move constructor** (lines 50–54) declares its parameter as an **rvalue reference (&&)** to indicate that its `MyArray` argument must be a temporary object. The member initializer list moves members `m_size` and `m_ptr` from the argument into the object being constructed.

[Click here to view code image](#)

```
49 // move constructor: must receive an rvalue
50 MyArray::MyArray(MyArray&& original) noexcept
51     : m_size{std::exchange(original.m_size,
52                         m_ptr{std::move(original.m_ptr)})} { // ...
53     cout << "MyArray move constructor\n";
54 }
55
```



SE A Recall from [Section 11.6.2](#) that the only valid operations on a moved-from object are assigning another object to it or destroying it. **When moving resources from an object, it should be left in a state that allows it to be properly destructed. Also, it should no longer refer to the resources that were moved to the new object.**²³ To accomplish this for the `m_size` member, line 51

²³. “C.64: A move operation should move and leave its source in a valid state.” Accessed February 26, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-semantic>.

- calls the standard library **exchange function** (header `<utility>`), which sets its first argument (`original.m_size`) to its second argument’s value (0) and returns its first argument’s original value, then
- initializes the new object’s `m_size` data member with the value that `exchange` returns.

When a `unique_ptr` is **move constructed**, as in line 52, its **move constructor** transfers ownership of the source `unique_ptr`'s dynamic memory to the target `unique_ptr` and sets the source `unique_ptr` to `nullptr`.²⁴ If your class manages raw pointers, you'd have to explicitly set the source pointer to `nullptr`—or use `exchange`, similar to line 51.

24. “`std::unique_ptr<T,Deleter>::unique_ptr`.” Accessed February 26, 2021. https://en.cppreference.com/w/cpp/memory/unique_ptr/unique_ptr.

Moving Does Not Move Anything

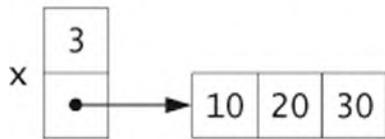
Though we said the move constructor “moves the `m_size` and `m_ptr` members from the argument object into the object being constructed,” it does not actually move anything.²⁵

25. Topher Winward, “C++ moves for people who don’t know or care what rvalues are,” January 17, 2019. Accessed March 3, 2021. <https://medium.com/@winwardo/c-moves-for-people-whodont-know-or-care-what-rvalues-are-%EF%B8%8F-56ee122dda7>.

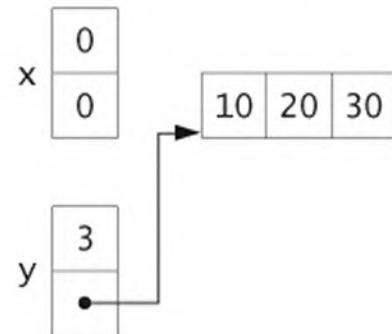
- For fundamental types like `size_t`, which is simply an unsigned integer, **the value is copied** from the source object's member to the new object's member.
- For a raw pointer, the address stored in the source object's pointer is copied to the new object's pointer.
- For an object, the object's move constructor is called. A `unique_ptr`'s move constructor **transfers ownership** of the dynamic memory by copying the dynamically allocated memory's address from the source `unique_ptr`'s underlying raw pointer into the new object's underlying raw pointer, then assigning `nullptr` to the source `unique_ptr` to indicate it no longer manages any data.

The diagram below shows the concept of moving a source object `x` with a raw pointer member into a new object `y`. Note that both members of `x` are 0 after the move—0 in a pointer member represents a null pointer.

Before x is moved into y



After x is moved into y



Class MyArray's Move Assignment Operator (=)

The **move assignment operator (=)** (lines 57–67) defines an **rvalue reference (&&)** parameter to indicate that its MyArray argument's resources should be moved (not copied). Line 60 tests for **self-assignment** in which a MyArray object is being **assigned to itself**.²⁶ When this is equal to the right operand's address, the same object is on both sides of the assignment, so there's no need to move anything.

26. “C.65: Make move assignment safe for self-assignment.” Accessed March 3, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self>.

[Click here to view code image](#)

```
56 // move assignment operator
57 MyArray& MyArray::operator=(MyArray&& right
58     cout << "MyArray move assignment operato
59
60     if (this != &right) { // avoid self-assi
61         // move right's data into this MyArra
62         m_size = std::exchange(right.m_size,
63         m_ptr = std::move(right.m_ptr);
64     }
65
66     return *this; // enables x = y = z, for
67 }
68
```

If it is not a **self-assignment**, lines 62–63:

- move `right.m_size` into the target `MyArray`'s `m_size` by calling `exchange`—this sets `right.m_size` to 0 and returns its original value, which we use to set the target `MyArray`'s `m_size` member, and
- move `right.m_ptr` into the target `MyArray`'s `m_ptr`.

As in the **move constructor**, when a `unique_ptr` is **move assigned**, ownership of its dynamic memory transfers to the new `unique_ptr`, and the **move assignment operator** sets the original `unique_ptr` to `nullptr`. Regardless of whether this is a self-assignment, the member function returns the current object (`*this`), which enables cascaded `MyArray` assignments such as `x=y=z`.

11 Move Operations Should Be `noexcept`

 **Move constructors and move assignment operators should never throw exceptions.** They do not acquire any new resources—they simply *move* existing ones. For this reason, the C++ Core Guidelines recommend declaring **move constructors** and **move assignment operators** `noexcept`.²⁷ This also is a requirement to be able to use your class's move capabilities with the standard library's containers like `vector`.

27. “C.66: Make move operations noexcept.” Accessed February 25, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept>.

11.6.8 Destructor

Line 27 of Fig. 11.4

```
~MyArray(); // destructor
```

 declares the class's **destructor** (defined in lines 71–73), which is invoked when a `MyArray` object goes out of scope. This will automatically call `m_ptr`'s destructor, which will **release the dynamically allocated int array created when the MyArray object was constructed**. The C++ Core Guidelines indicate that it's poor design if destructors throw exceptions. For this reason, they recommend declaring destructors `noexcept`.²⁸

However, unless your class is derived from a base class with a destructor that's declared `noexcept` (`false`), the compiler implicitly declares the destructor `noexcept` by default.

28. “C.37: Make destructors `noexcept`.” Accessed February 25, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-noexcept>.

[Click here to view code image](#)

```
69 // destructor: This could be compiler-generated
70 // we could output when each MyArray is destroyed
71 MyArray::~MyArray() {
72     cout << "MyArray destructor\n";
73 }
74
```



11.6.9 `toString` and `size` Functions

Line 29 in Fig. 11.4:

[Click here to view code image](#)

```
size_t size() const noexcept {return m_size;} //
```



defines an inline `size` member function, which returns a `MyArray`'s number of elements.

Line 30 in Fig. 11.4:

[Click here to view code image](#)

```
std::string toString() const; // create string representation
```



declares a `toString` member function (defined in lines 76–89), which returns the string representation of a `MyArray`'s contents. Function `toString` uses an `ostringstream` (introduced in Section 8.17) to build a string containing the `MyArray`'s element values enclosed in braces `{ }` and separating each `int` from the next by a comma and a space.

[Click here to view code image](#)

```
75 // return a string representation of a MyAr
76 string MyArray::toString() const {
77     const span<const int> items{m_ptr.get(),
78     ostringstream output;
79     output << "{";
80
81     // insert each item in the dynamic array
82     for (size_t count{0}; const auto& item :
83         ++count;
84         output << item << (count < m_size ? " "
85     }
86
87     output << "}";
88     return output.str();
89 }
90
```



11.6.10 Overloading the Equality (==) and Inequality (!=) Operators

Line 33 of Fig. 11.4:

[Click here to view code image](#)

```
bool operator==(const MyArray& right) const noexcept
```



declares the **overloaded equality operator (==)**. Comparisons should not throw exceptions, so they should be declared noexcept.²⁹

²⁹. “C.86: Make == symmetric with respect of operand types and noexcept.” Accessed February 25, 2021.

[https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq.](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq)

When the compiler sees an expression like `ints1 == ints2`, it invokes this overloaded operator with the call

```
ints1.operator==(ints2)
```

Member function `operator==` (defined in lines 93–102) operates as follows:

- Line 95 creates the span `lhs` representing the dynamically allocated `int` array in the left-hand-side operand (`ints1`).
- Line 96 creates the span `rhs` representing the dynamically allocated `int` array in the right-hand-side operand (`ints2`).
- Line 97 uses the standard library algorithm `equal` (from header `<algorithm>`) to compare corresponding elements from each spans. The first two arguments specify the `lhs` object's range of elements. The last two specify the `rhs` object's range of elements. If the `lhs` and `rhs` objects have different lengths or if any pair of corresponding elements differ, `equal` returns `false`. If every pair of elements is equal, `equal` returns `true`.

[Click here to view code image](#)

```
91 // determine if two MyArrays are equal and
92 // return true, otherwise return false
93 bool MyArray::operator==(const MyArray& right)
94     // compare corresponding elements of both arrays
95     const span<const int> lhs{m_ptr.get(), s};
96     const span<const int> rhs{right.m_ptr.get(), r};
97     return equal(begin(lhs), end(lhs), begin(rhs), end(rhs));
98 }
99 }
```



Complier Generates the `!=` Operator

20 As of C++20, **the compiler autogenerated a `!=` operator function for you if you provide the `==` operator for your type**. Prior to C++20, if your class required a custom **overloaded inequality operator** (`!=`) operator, you'd define `!=` to call the `==` operator function and return the opposite

result.

Defining Comparison Operators as Non-Member Functions

  Each class we've defined so far, including `MyArray`, declared its single-argument constructor(s) explicit to prevent implicit conversions, as recommended by the C++ Core Guide-line "C.164: Avoid implicit conversion operators."³⁰ You may also come across the C++ Core Guideline "C.86: Make `==` symmetric with respect to operand types and `noexcept`,"³¹ which recommends making comparison operators non-member functions **if your class supports implicitly converting objects of other types to your class's type, or vice versa**. This guideline applies to all comparison operators, but we'll discuss `==` here, as it's the only comparison operator defined in our `MyArray` class.

30. "C.164: Avoid implicit conversion operators," Accessed April 9, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#R0-conversion>.

31. "C.86: Make `==` symmetric with respect to operand types and `noexcept`." Accessed April 9, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq>.

Assume `ints` is a `MyArray` and `other` is an object of class `OtherType`, which is **implicitly convertible to a `MyArray`**. To satisfy Core Guideline C.86, we'd define `operator==` as a non-member function with the prototype:

[Click here to view code image](#)

```
bool operator==(const MyArray& left, const MyArra:
```



This would allow the following mixed-type expressions:

```
ints == other
```

or

```
other == ints
```

 There is no `operator==` definition that provides parameters

exactly matching operands of types `MyArray` and `OtherType` or `OtherType` and `MyArray`. However, **C++ allows one user-defined conversion per expression.** So, if `OtherType` objects can be implicitly converted to `MyArray` objects, the compiler will convert `other` to a `MyArray`, then call the `operator==` that receives two `MyArrays`.

C++ follows a complex set of overload-resolution rules to determine which function to call for each operator expression.³² If `operator==` is a `MyArray` member function, **the left operand must be a `MyArray`.** C++ will not implicitly convert `other` to a `MyArray` to call the member function, so the expression

³² “Overload resolution.” Accessed April 10, 2021. https://en.cppreference.com/w/cpp/language/overload_resolution.

```
other == ints
```

with an `OtherType` object on the left causes a compilation error. This might confuse programmers who'd expect this expression to compile, which is why Core Guideline C.86 recommends using a non-member `operator==` function.

11.6.11 Overloading the Subscript ([]) Operator

Lines 36 and 39 of Fig. 11.4

[Click here to view code image](#)

```
int& operator[](size_t index);  
int operator[](size_t index) const;
```

declare **overloaded subscript operators** (defined in lines 102–109 and 113–120). When the compiler sees an expression like `ints1[5]`, it invokes the appropriate **overloaded operator [] member function** by generating the call

```
ints1.operator[](5)
```

The compiler calls the `const` version of `operator[]` (lines 113–120) when the **subscript operator** is used on a `const MyArray` object. For example, if you pass a `MyArray` to a function that receives the `MyArray` as a `const MyArray&` named `z`, then the **const version of operator[]**

is required to execute a statement such as

```
cout << z[3];
```

Remember that when an object is `const`, a program can invoke only the object's `const` member functions.

[Click here to view code image](#)

```
100   // overloaded subscript operator for non-const objects
101   // reference return creates a modifiable lvalue
102   int& MyArray::operator[](size_t index) {
103       // check for index out-of-range error
104       if (index >= m_size) {
105           throw out_of_range("Index out of range");
106       }
107
108       return m_ptr[index]; // reference return
109   }
110
111   // overloaded subscript operator for const objects
112   // const reference return creates a non-modifiable lvalue
113   const int& MyArray::operator[](size_t index) const {
114       // check for subscript out-of-range error
115       if (index >= m_size) {
116           throw out_of_range("Index out of range");
117       }
118
119       return m_ptr[index]; // returns copy of value
120   }
121
```



Each `operator[]` definition determines whether argument `index` is in range. If not, it throws an `out_of_range exception` (header `<stdexcept>`). If `index` is in range, the **non-const version of `operator[]`** returns the appropriate `MyArray` element as a reference. This

may be used as a modifiable *lvalue* on an assignment's left side to modify an array element. The **const version of operator[]** returns a **const** reference to the appropriate array element.

The subscript operators used in lines 108 and 119 belong to class `unique_ptr`. When a `unique_ptr` manages a dynamically allocated array, `unique_ptr`'s overloaded `[]` operator enables you to access the array's elements.

11.6.12 Overloading the Unary `bool` Conversion Operator

You can define your own **conversion operators** for converting between types—these are also called **overloaded cast operators**. Line 42 of Fig. 11.4

[Click here to view code image](#)

```
explicit operator bool() const noexcept {return s:
```



defines an **inline** overloaded operator `bool` that converts a `MyArray` object to the `bool` value `true` if the `MyArray` is not empty or `false` if it is. Like single-argument constructors, we declared this operator `explicit` to prevent the compiler from using it for implicit conversions (we say more about this in [Section 11.9](#)). **Overloaded conversion operators do not specify a return type to the left of the `operator` keyword. The return type is the conversion operator's type—`bool` in this case.**

In [Fig. 11.3](#), line 99 used the `MyArray` `ints5` as an `if` statement's condition to determine whether it contained elements. In that case, **C++ called this operator `bool` function to perform a contextual conversion of the `ints5` object to a `bool` value for use as a condition.** You also may call this function explicitly using an expression like:

```
static_cast<bool>(ints5)
```

We say more about converting between types in [Section 11.8](#).

11.6.13 Overloading the Preincrement Operator

You can overload the prefix and postfix increment and decrement operators. The concepts we show here and in [Section 11.6.14](#) for `++` also apply to the `--` operators. [Section 11.6.14](#) shows how the compiler distinguishes between

the prefix and postfix versions.

Line 45 of Fig. 11.4

```
MyArray& operator++();
```

declares MyArray's unary **overloaded preincrement operator (++)**. When the compiler sees an expression like `++ints4`, it invokes MyArray's overloaded preincrement operator (++) function by generating the call

```
ints4.operator++()
```

This invokes the function's definition (lines 123–128), which adds one to each element by:

- creating the span `items` to represent the dynamically allocated `int` array, then
- using the standard library's **for_each** algorithm to call a function that performs a task once for each element of the span.

[Click here to view code image](#)

```
122 // preincrement every element, then return
123 MyArray& MyArray::operator++() {
124     // use a span and for_each to increment
125     const span<int> items{m_ptr.get(), m_si
126     for_each(begin(items), end(items), [](auto&
127         return *this;
128     }
129 }
```



Like the `copy` algorithm, `for_each`'s first two arguments represent the range of elements to process. The third argument is a function that receives one argument and performs a task with it. In this case, we specify a **lambda expression** that is called once for each element in the range. As `for_each` iterates internally through the span's elements, it passes the current element as the lambda's argument (`item`). The lambda then performs a task using that value. This lambda's argument is a non-const reference (`auto&`), so

the expression `++item` in the lambda's body modifies the original element in the `MyArray`.

The operator returns a reference to the `MyArray` object it just incremented. This enables a preincremented `MyArray` object to be used as an *lvalue*, which is how the builtin prefix increment operator works for fundamental types.

11.6.14 Overloading the Postincrement Operator

 Overloading the postfix increment operator presents a challenge. The compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions. By convention, when the compiler sees a postincrement expression like `ints4++`, it generates the member-function call

```
ints4.operator++(0)
```

The argument `0` is strictly a **dummy value** that the compiler uses to distinguish between the prefix and postfix increment operator functions. The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

Line 48 of Fig. 11.4:

```
MyArray operator++(int);
```

declares `MyArray`'s unary **overloaded postincrement operator (++)** with the `int` parameter that receives the dummy value `0`. The parameter is not used, so it's declared without a parameter name. To emulate the effect of the postincrement, we must return an **unincremented copy** of the `MyArray` object. So, the function's definition (lines 131–135):

- uses the `MyArray` **copy constructor** to make a local copy of the original `MyArray`,
- calls the preincrement operator to add one to every element of the `MyArray`³³, then

³³. Herb Sutter, “GotW #2 Solution: Temporary Objects,” May 13, 2013. Accessed February 26, 2021. <https://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/>.

- **Perf**  returns the unincremented local copy of the `MyArray` by *value*—this is another case in which compilers can use the **named return value optimization (NRVO)**.

Perf  The extra local object created by the postfix increment (or decrement) operator can result in a performance problem, especially if the operator is used in a loop. For this reason, you should **prefer the prefix increment and decrement operators**.

[Click here to view code image](#)

```

130    // postincrement every element, and return
131    MyArray MyArray::operator++(int) {
132        MyArray temp(*this);
133        ++(*this); // call preincrement operator
134        return temp; // return the temporary copy
135    }
136

```



11.6.15 Overloading the Addition Assignment Operator (+=)

Line 51 of Fig. 11.4

[Click here to view code image](#)

```
MyArray& operator+=(int value);
```

declares `MyArray`'s **overloaded addition assignment operator (+=)**, which adds a value to every element of a `MyArray`, then **returns a reference to the modified object** to enable cascaded calls. Like the preincrement operator, we use a span and the standard library function `for_each` to process every element in the `MyArray`. In this case, the lambda we pass as `for_each`'s last argument (line 142) uses the `operator+=` function's `value` parameter in its body. The **lambda introducer** `[value]` specifies that the compiler should allow `value` to be used in the lambda's body—this is known as **capturing** the variable. We'll say more about capturing lambdas in Chapter 14.

[Click here to view code image](#)

```
137 // add value to every element, then return
138 MyArray& MyArray::operator+=(int value) {
139     // use a span and for_each to increment
140     const span<int> items{m_ptr.get(), m_si
141     for_each(begin(items), end(items),
142             [value](auto& item) {item += value;})
143     return *this;
144 }
145
```



11.6.16 Overloading the Binary Stream Extraction (>>) and Stream Insertion (<<) Operators

You can input and output fundamental-type data using the **stream extraction operator (>>)** and the **stream insertion operator (<<)**. The C++ standard library overloads these operators for each fundamental type, including pointers and `char*` strings. You also can overload these to perform input and output for custom types.

Line 10 of [Fig. 11.4](#)

[Click here to view code image](#)

```
friend std::istream& operator>>(std::istream& in,
```



and line 58 of [Fig. 11.4](#)

[Click here to view code image](#)

```
std::ostream& operator<<(std::ostream& out, const
```



Perf  declare the non-member **overloaded stream-extraction operator (>>)** and **overloaded stream-insertion operator (<<)**. We declared `operator>>` in the class as a friend because it will access a `MyArray`'s private data directly for performance. The `operator<<` is not declared

as a friend. As you'll see, it calls `MyArray`'s `toString` member function to get a `MyArray`'s string representation then outputs it.

Implementing the Stream Extraction Operator

The **overloaded stream-extraction operator (`>>`)** (lines 147–156) takes as arguments an `istream` reference and a `MyArray` reference. It returns the `istream` reference argument to enable **cascaded inputs**, like

```
cin >> ints1 >> ints2;
```

When the compiler sees an expression like `cin >> ints1`, it invokes **non-member function operator`>>`** with the call

```
operator>>(cin, ints1)
```

We'll say why this needs to be a **non-member function** momentarily. When this call completes, it returns a reference to `cin`, which would then be used in the `cin` statement above to input values into `ints2`. The function creates a span (line 149) representing `MyArray`'s dynamically allocated `int` array. Then lines 151–153 iterate through the span's elements, reading one value at a time from the input stream and placing the value in the corresponding element of the dynamically allocated `int` array.

[Click here to view code image](#)

```
146 // overloaded input operator for class MyA
147 // inputs values for entire MyArray
148 istream& operator>>(istream& in, MyArray&
149     span<int> items{a.m_ptr.get(), a.m_size
150
151     for (auto& item : items) {
152         in >> item;
153     }
154
155     return in; // enables cin >> x >> y;
156 }
157
```



Implementing the Stream Insertion Operator

The **overloaded stream insertion function (<<)** (lines 159–162) receives an `ostream` reference and a `const MyArray` reference as arguments and returns an `ostream` reference. The function calls `MyArray`'s `toString` member function then outputs the resulting string. The function returns the `ostream` reference argument to enable **cascaded output statements**, like

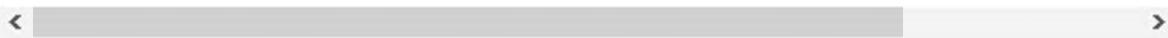
```
cout << ints1 << ints2;
```

When the compiler sees an expression like `cout << ints1`, it invokes **non-member function operator<<** with the call

```
operator<<(cout, ints1)
```

[Click here to view code image](#)

```
158 // overloaded output operator for class My
159 ostream& operator<<(ostream& out, const My
160     out << a.toString();
161     return out; // enables cout << x << y;
162 }
163
```



Why `operator>>` and `operator<<` Must Be Nonmember Functions

The `operator>>` and `operator<<` functions are defined as **non-member functions**, so we can specify their operands' order in each function's parameter list. In a binary overloaded operator implemented as a **non-member function**, the first parameter is the left operand, and the second is the right operand.

For the operators `>>` and `<<`, the `MyArray` object should be each operator's **right operand**, so you can use them the way C++ programmers expect, as in the statements:

```
cin >> ints4;
```

```
cout << ints4;
```

If we defined these functions as `MyArray` member functions, programmers would have to write the following awkward statements to input or output `MyArray` objects:

```
ints4 >> cin;  
ints4 << cout;
```

Such statements would be confusing and, in some cases, would lead to compilation errors. Programmers are familiar with `cin` and `cout` always appearing to the *left* of these operators.

SE A Overloaded binary operators may be member functions only for the class of the operator's *left* operand. For `operator>>` and `operator<<` to be member functions, we'd have to modify the standard library classes `istream` and `ostream`, which is not allowed.

Choosing Member vs. Non-Member Functions

Overloaded operator functions, and functions in general, can be:

- member functions with direct access to the class's internal implementation details,
- friend functions with direct access to the class's internal implementation details, or
- non-member, non-friend functions—often called **free functions**—that interact with objects of the class through its `public` interface.

CG The C++ Core Guidelines say a function should be a member only if it needs direct access to the class's internal implementation details, such as its `private` data.³⁴ Another reason to

³⁴. “C.4: Make a function a member only if it needs direct access to the representation of a class.” Accessed March 6, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-member>.

use non-member functions is to define **commutative operators**. Consider a class `HugeInt` for

arbitrary-sized integers. With a `HugeInt` named `bigInt`, we might write

expressions like:

```
bigInt + 7  
7 + bigInt
```

Each would sum an `int` value and a `HugeInt`. Like the built-in `+` operator for fundamental types, each would produce temporary `HugeInt` containing the sum. To support these expressions, you define two versions of `operator+`, typically as non-member `friend` functions:

[Click here to view code image](#)

```
friend HugeInt operator+(const HugeInt& left, int right);  
friend HugeInt operator+(int left, const HugeInt& right);
```

◀ ▶

To avoid code duplication, the second function typically would call the first.

11.6.17 friend Function `swap`

Line 13 of [Fig. 11.4](#)

[Click here to view code image](#)

```
friend void swap(MyArray& a, MyArray& b) noexcept;
```

◀ ▶

CG (C++ Guidelines) declares the `swap` function used by the **copy assignment operator** to implement the **copy-and-swap idiom**. This function is declared `noexcept`—exchanging the contents of two existing objects does not allocate new resources, so it should not fail.³⁵ The function (lines 165–168) receives two `MyArrays`. It uses the **standard library swap function** to exchange the contents of each object’s `m_size` members. It uses the `unique_ptr`’s **swap member function** to exchange the contents of the each object’s `m_ptr` members.

35. “C.84: A `swap` function may not fail.” Accessed February 25, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-swap-fail>.

[Click here to view code image](#)

```
164 // swap function used to implement copy-and-swap
165 void swap(MyArray& a, MyArray& b) noexcept
166     std::swap(a.m_size, b.m_size); // swap
167     a.m_ptr.swap(b.m_ptr); // swap using uniform swap
168 }
```



20 11.7 C++20 Three-Way Comparison Operator ($\langle=\rangle$)

You'll often compare objects of your custom class types. For example, to sort objects into ascending or descending order using the standard library's `sort` function (Section 6.12; more details in Chapter 14), the objects must be comparable. To support comparisons, you can overload the equality (`==` and `!=`) and relational (`<`, `<=`, `>` and `<=`) operators for your classes. A common practice is to define the `<` and `==` operator functions, then define `!=`, `<=`, `>` and `<=` operators in terms of `<` and `==`. For instance, for a class `Time` that represents the time of day, its `<=` operator could be implemented as an inline member function in terms of the `<` operator:

[Click here to view code image](#)

```
bool operator<=(const Time& right) const {
    return !(right < *this);
}
```

This leads to lots of “boilerplate” code in which the only difference in the definitions of the overloaded `!=`, `<=`, `>` and `<=` operators among classes is the argument type.

For most types, the compiler can handle the comparison operators for you via the compiler-generated default implementation of C++20's **three-way comparison operator ($\langle=\rangle$)**,^{36,37,38} which is also referred to as the spaceship operator³⁹ and requires the header `<compare>`. Figure 11.6 demonstrates `<=` for a class `Time` (lines 9–24), which contains:

³⁶ “C++ Russia 2018: Herb Sutter, New in C++20: The spaceship operator,” YouTube Video, June 25, 2018, <https://www.youtube.com/watch?v=ULkwKsag0Yk>.

37. Sy Brand, “Spaceship Operator,” August 23, 2018. Accessed March 6, 2021. <https://blog.tartanllama.xyz/spaceship-operator/>.

38. Cameron DaCamara, “Simplify Your Code With Rocket Science: C++20’s Spaceship Operator,” June 27, 2019. Accessed March 6, 2021. <https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/>.

39. “Spaceship operator” was coined by Randal L. Schwartz when he was teaching the same operator in a Perl programming course—the operator reminded him of a spaceship in an early video game. <https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftItp14/m/jcpli=1>.

- a constructor (lines 11–12) to initialize its private data members (lines 21–23),
- a `toString` function (lines 14–16) to create a `Time`’s string representation, and
- a **defaulted definition of the overloaded `<=>` operator** (line 19) that the compiler will generate.

SE  **The `default` compiler-generated `operator<=>` works for any class containing data members that all support the equality and relational operators.** Also, for classes containing built-in arrays as data members, the compiler applies the overloaded `operator<=>` element-by-element as it compares two objects of the arrays’ enclosing class type.

[Click here to view code image](#)

```
1 // fig11_06.cpp
2 // C++20 three-way comparison (spaceship) operator
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include "fmt/format.h"
7 using namespace std;
8
9 class Time {
10 public:
11     Time(int hr, int min, int sec) noexcept
12         : m_hr{hr}, m_min{min}, m_sec{sec} {}
```

```

13
14     string toString() const {
15         return fmt::format("hr={}, min={}, sec={}", hr, min, sec);
16     }
17
18     // <=> operator automatically supports equality comparison
19     auto operator<=>(const Time& t) const noexcept {
20     private:
21         int m_hr{0};
22         int m_min{0};
23         int m_sec{0};
24     };
25
26     int main() {
27         const Time t1(12, 15, 30);
28         const Time t2(12, 15, 30);
29         const Time t3(6, 30, 0);
30
31         cout << fmt::format("t1: {}\nt2: {}\nt3: {}",
32                             t1.toString(), t2.toString(),
33                             t3.toString());

```

```
t1: hr=12, min=15, sec=30
t2: hr=12, min=15, sec=30
t3: hr=6, min=30, sec=0
```

Fig. 11.6 | C++20 three-way comparison (spaceship) operator.

In `main`, lines 27–29 create three `Time` objects that we'll use in various comparisons, then display their `string` representations. `Time` objects `t1` and `t2` both represent 12:15:30 PM, and `t3` represents 6:30:00 AM. The rest of this program is broken into smaller pieces for discussion purposes.

With `<=>`, the Compiler Supports All the Comparison Operators

When you let the compiler generate the overloaded three-way comparison

operator (`<=>`) for you, your class supports all the relational and equality operators, which we demonstrate in lines 35–47. In each case, the compiler rewrites an expression like

```
t1 == t2
```

into an expression that uses the `<=>` operator, as in

```
(t1 <=> t2) == 0
```

The expression `t1 <=> t2` evaluates to 0 if the objects are equal, a negative value if `t1` is less than `t2` and a positive value if `t1` is greater than `t2`. As you can see, lines 35–47 compiled and produced correct outputs, even though class `Time` did not define any equality or relational operators.

[Click here to view code image](#)

```
34 // using the equality and relational operators
35 cout << fmt::format("t1 == t2: {}\n", t1 == t2);
36 cout << fmt::format("t1 != t2: {}\n", t1 != t2);
37 cout << fmt::format("t1 < t2: {}\n", t1 < t2);
38 cout << fmt::format("t1 <= t2: {}\n", t1 <= t2);
39 cout << fmt::format("t1 > t2: {}\n", t1 > t2);
40 cout << fmt::format("t1 >= t2: {}\n\n", t1 >= t2);
41
42 cout << fmt::format("t1 == t3: {}\n", t1 == t3);
43 cout << fmt::format("t1 != t3: {}\n", t1 != t3);
44 cout << fmt::format("t1 < t3: {}\n", t1 < t3);
45 cout << fmt::format("t1 <= t3: {}\n", t1 <= t3);
46 cout << fmt::format("t1 > t3: {}\n", t1 > t3);
47 cout << fmt::format("t1 >= t3: {}\n\n", t1 >= t3);
48
```

◀ ▶

```
t1 == t2: true
t1 != t2: false
t1 < t2: false
t1 <= t2: true
```

```
t1 > t2: false  
t1 >= t2: true  
  
t1 == t3: false  
t1 != t3: true  
t1 < t3: false  
t1 <= t3: false  
t1 > t3: true  
t1 >= t3: true
```

Using `<=>` Explicitly

You also can use `<=>` in expressions. An `<=>` expression's result is not convertible to `bool`, so you must compare it to 0 to use `<=>` in a condition, as shown in lines 50, 54 and 58.

[Click here to view code image](#)

```
49     // using <=> to perform comparisons  
50     if ((t1 <=> t2) == 0) {  
51         cout << "t1 is equal to t2\n";  
52     }  
53  
54     if ((t1 <=> t3) > 0) {  
55         cout << "t1 is greater than t3\n";  
56     }  
57  
58     if ((t3 <=> t1) < 0) {  
59         cout << "t3 is less than t1\n";  
60     }  
61 }
```

```
t1 is equal to t2  
t1 is greater than t3  
t3 is less than t1
```

In Chapter 19, Other Topics and a Look Toward the Future of C++, we'll discuss overloading this operator for more complex types, like our `MyArray` class that contains dynamically allocated memory.

11.8 Converting Between Types

Most programs process information of many types. Sometimes all the operations “stay within a type.” For example, adding an `int` to an `int` produces an `int`. It’s often necessary, however, to convert data of one type to data of another type. This can happen in assignments, calculations, passing values to functions and returning values from functions. The compiler knows how to perform certain conversions among fundamental types. You can use cast operators to force conversions among fundamental types.

But what about user-defined types? The compiler does not know how to convert among user-defined types or between user-defined types and fundamental types. You must specify how to do this. Such conversions can be performed with **conversion constructors**. These constructors are called with one argument (we refer to these as **single-argument constructors**). Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

Conversion Operators

A **conversion operator** (also called a **cast operator**) also can convert an object of a class to another type. Such a conversion operator must be a *nonstatic member function*. In the `MyArray` case study, we implemented an overloaded conversion operator that converted a `MyArray` to a `bool` value to determine whether the `MyArray` contained elements.

Implicit Calls to Cast Operators and Conversion Constructors

 A feature of cast operators and conversion constructors is that the compiler can call them *implicitly* to create objects. For example, you saw that when an object of our class `MyArray` appears in a program where a `bool` is expected, such as

[Click here to view code image](#)

```
if (ints1) { // if expects a condition
```

}

the compiler can call the overloaded cast-operator function operator bool to convert the object into a bool and use the resulting bool in the expression.

SE  When a conversion constructor or conversion operator is used to perform an implicit conversion, C++ can apply only one implicit constructor or operator function call (i.e., a single user-defined conversion) per expression to try to match the needs of that expression. The compiler will not satisfy an expression's needs by performing a series of implicit, user-defined conversions.

11.9 **explicit** Constructors and Conversion Operators

SE  Recall that we've been declaring as explicit every constructor that can be called with one argument, including multiparameter constructors for which we specify default arguments. Except for copy and move constructors, any constructor that can be called with a *single argument* and is *not* declared **explicit** can be used by the compiler to perform an *implicit conversion*. The constructor's argument is converted to an object of the class in which the constructor is defined. The conversion is automatic—a cast is not required.

In some situations, implicit conversions are undesirable or error-prone. For example, our MyArray class defines a constructor that takes a single size_t argument. The intent of this constructor is to create a MyArray object containing a specified number of elements. However, if this constructor were not declared **explicit** it could be misused by the compiler to perform an *implicit conversion*.

Err  Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in execution-time logic errors or ambiguous expressions that generate compilation errors.

Accidentally Using a Single-Argument Constructor as a Conversion

Constructor

The program (Fig. 11.7) uses Section 11.6's MyArray class to demonstrate an improper implicit conversion. To allow this implicit conversion, we removed the `explicit` keyword from line 16 in `MyArray.h` (Fig. 11.4).

[Click here to view code image](#)

```
1 // fig11_07.cpp
2 // Single-argument constructors and implicit conversions
3 #include <iostream>
4 #include "MyArray.h"
5 using namespace std;
6
7 void outputArray(const MyArray&); // prototype
8
9 int main() {
10     MyArray ints1(7); // 7-element MyArray
11     outputArray(ints1); // output MyArray in its original form
12     outputArray(3); // convert 3 to a MyArray
13 }
14
15 // print MyArray contents
16 void outputArray(const MyArray& arrayToOutput) {
17     cout << "The MyArray received has " << arrayToOutput.size()
18             << " elements. The contents are: " << arrayToOutput;
19 }
```



```
MyArray(size_t) constructor
The MyArray received has 7 elements. The content is { 1, 2, 3, 4, 5, 6, 7 }
MyArray(size_t) constructor
The MyArray received has 3 elements. The content is { 1, 2, 3 }
MyArray destructor
MyArray destructor
```

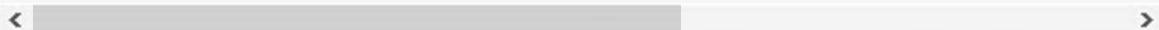


Fig. 11.7 | Single-argument constructors and implicit conversions.

Line 10 in `main` (Fig. 11.7) instantiates `MyArray` object `ints1` and calls the *single-argument constructor* with the value 7 to specify `ints1`'s number of elements. Recall that the `MyArray` constructor that receives a `size_t` argument initializes all the `MyArray` elements to 0. Line 11 calls function `outputArray` (defined in lines 16–19), which receives as its argument a `const MyArray&`. The function outputs its argument's number of elements and contents. In this case, the `MyArray`'s size is 7, so `outputArray` displays seven 0s.

Line 12 calls `outputArray` with the value 3 as an argument. This program does *not* contain an `outputArray` function that takes an `int` argument. So, the compiler determines whether the argument 3 can be converted to an `MyArray` object. Because class `MyArray` provides a constructor with one `size_t` argument (which can receive an `int`) and that constructor is not declared `explicit`, the compiler assumes the constructor is a **conversion constructor** and uses it to convert the argument 3 into a temporary `MyArray` object containing three elements. Then, the compiler passes this temporary `MyArray` object to function `outputArray`, which displays the temporary `MyArray`'s size and contents. Thus, even though we do not *explicitly* provide an `outputArray` function that receives an `int`, the compiler can compile line 12. The output shows the contents of the three-element `MyArray` containing 0s.

Preventing Implicit Conversions with Single-Argument Constructors

 **The reason we've been declaring every single-argument constructor preceded by the keyword `explicit` is to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that's declared `explicit` cannot be used in an `implicit` conversion.** Our next program uses the original `MyArray` version from Section 11.6, which included the keyword `explicit` in the declaration of its **single-argument constructor** that receives a `size_t`:

[Click here to view code image](#)

```
explicit MyArray(size_t size);
```

Figure 11.8 presents a slightly modified version of the program in Fig. 11.7. When this program in Fig. 11.8 is compiled, the compiler produces an error message, such as

[Click here to view code image](#)

```
error: invalid initialization of reference of type
      from expression of type 'int'
```

on g++, indicating that the integer value passed to outputArray in line 12 *cannot* be converted to a `const MyArray&`. Line 13 demonstrates how the explicit constructor can be used explicitly to create a temporary `MyArray` of 3 elements and pass it to `outputArray`.

[Click here to view code image](#)

```
1 // fig11_08.cpp
2 // Demonstrating an explicit constructor.
3 #include <iostream>
4 #include "MyArray.h"
5 using namespace std;
6
7 void outputArray(const MyArray&); // prototype
8
9 int main() {
10     MyArray ints1{7}; // 7-element MyArray
11     outputArray(ints1); // output MyArray in
12     outputArray(3); // convert 3 to a MyArra
13     outputArray(MyArray(3)); // explicit sin
14 }
15
16 // print MyArray contents
17 void outputArray(const MyArray& arrayToOutp
18     cout << "The MyArray received has " << a
19         << " elements. The contents are: " <<
20 }
```

Fig. 11.8 | Demonstrating an explicit constructor.

SE A 20 You should always use the `explicit` keyword on single-argument constructors unless they're intended to be used implicitly as conversion constructors. In this case, you should declare the single-argument constructor `explicit(false)`. This documents for your class's users that you wish to allow implicit conversions to be performed with that constructor. This new use of `explicit` was introduced in C++20.

11 C++11 `explicit` Conversion Operators

SE A 20 Just as you can declare single-argument constructors `explicit`, you can declare conversion operators `explicit`—or in C++20, `explicit(true)`—to prevent the compiler from using them to perform implicit conversions. For example, in class `MyArray`, the prototype

[Click here to view code image](#)

```
explicit operator bool() const noexcept;
```

declares the `bool` cast operator `explicit`, so you'd generally have to invoke it explicitly with `static_cast`, as in:

[Click here to view code image](#)

```
static_cast<bool>(myArrayObject)
```

As we showed in the `MyArray` case study, C++ can still perform contextual conversions using an `explicit bool` conversion operator to convert an object to a `bool` value in a condition.

11.10 Overloading the Function Call Operator ()

Overloading the **function-call operator ()** is powerful because functions can take an arbitrary number of comma-separated parameters. We demonstrate the overloaded function-call operator in a natural context in [Section 14.5](#).

11.11 Wrap-Up

This chapter demonstrated how to craft valuable classes, using operator overloading to enable C++’s existing operators to work with custom class objects. First, you used several `string`-class overloaded operators. Next, we presented operator-overloading fundamentals, including which operators can be overloaded and various rules and restrictions on operator overloading.

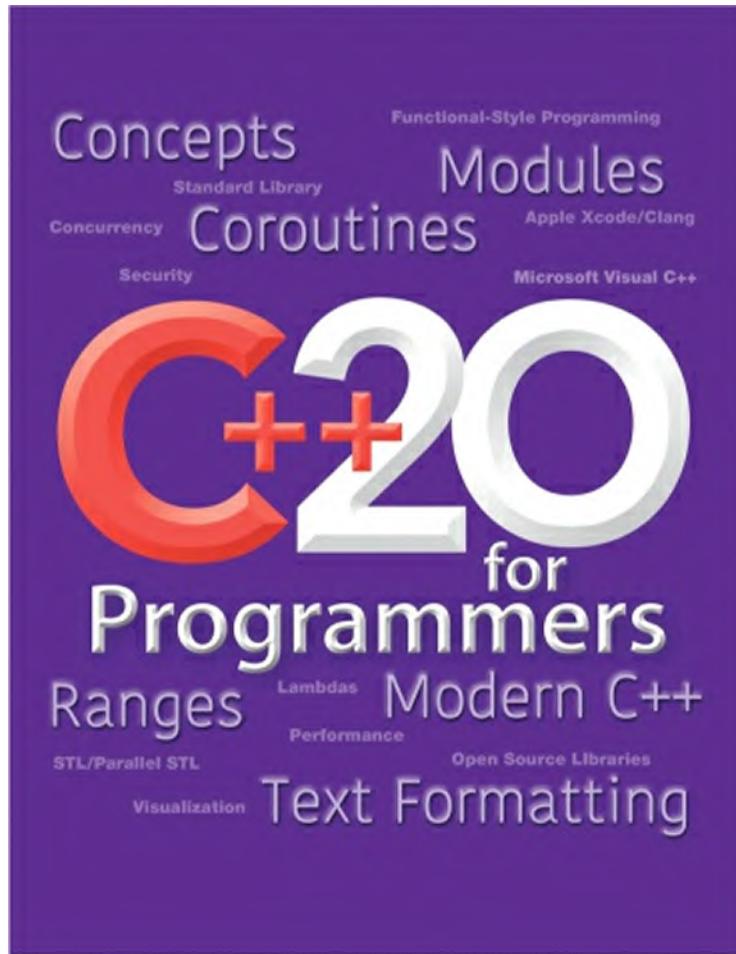
We introduced dynamic memory management with operators `new` and `delete`, which acquire and release the memory for objects and built-in, pointer-based arrays at runtime. We discussed the problems with using old-style `new` and `delete` statements, such as forgetting to use `delete` to release memory that’s no longer needed. We introduced RAII (Resource Acquisition Is Initialization) and demonstrated smart pointers. You saw how to dynamically allocate memory as you created a `unique_ptr` smart pointer object. The `unique_ptr` automatically released the memory when the object went out of scope, preventing a memory leak.

Next, we presented the chapter’s substantial capstone `MyArray` case study, which used overloaded operators and other capabilities to solve various problems with pointer-based arrays. We implemented the five special member functions typically defined in classes that manage their own resources—the `copy` constructor, `copy assignment` operator, `move constructor`, `move assignment` operator and `destructor`. We also discussed how to autogenerate these special member functions with `= default` and remove them with `= delete`. The class overloaded many operators and defined a conversion from `MyArray` to `bool` for determining whether a `MyArray` was empty or contained elements.

We introduced C++20’s new three-way comparison operator (`<= >`)—also called the spaceship operator. You saw that for some classes, the default compiler-generated `<= >` operator enables a class to support all six equality and relational operators without you having to explicitly overload them. We discussed in more detail converting between types, problems with implicit conversions defined by single-argument constructors and how to prevent those problems with the keyword `explicit`. Finally, we discussed overloading the function-call operator, `()`, which we’ll demonstrate in later chapters.

We've introduced some exception-handling fundamentals. The next chapter discusses exception handling in detail, so you can create more robust and fault-tolerant applications that can deal with problems and continue executing, or terminate gracefully. We'll show how to handle exceptions if operator `new` fails to allocate memory for an object. We'll also introduce several C++ standard library exception-handling classes and show how to create your own.

Chapter 12. Exceptions and a Look Forward to Contracts



Objectives

In this chapter, you'll:

- Understand the exception-handling flow of control with `try`, `catch` and `throw`.
- Provide exception guarantees for your code.
- Understand the standard library exception hierarchy.
- Define a custom exception class.

- Understand how stack unwinding enables exceptions not caught in one scope to be caught in an enclosing scope.
 - Handle new dynamic memory allocation failures.
 - Catch exceptions of any type with `catch(...)`.
 - Understand what happens with uncaught exceptions.
 - Understand which exceptions should not be handled and which cannot be handled.
 - Understand why some organizations disallow exceptions and the impact that can have on software-development efforts.
 - Understand the performance costs of exception handling.
 - Look ahead to how contracts can eliminate many use cases of exceptions, enabling more functions to be `noexcept`.
-

Outline

12.1 Introduction

12.2 Exception-Handling Flow of Control; Defining an Exception Class

12.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur

12.2.2 Demonstrating Exception Handling

12.2.3 Enclosing Code in a `try` Block

12.2.4 Defining a `catch` Handler for `DivideByZeroExceptions`

12.2.5 Termination Model of Exception Handling

12.2.6 Flow of Control When the User Enters a Nonzero Denominator

12.2.7 Flow of Control When the User Enters a Zero Denominator

12.3 Exception Safety Guarantees and `noexcept`

12.4 Rethrowing an Exception

12.5 Stack Unwinding and Uncaught Exceptions

12.6 When to Use Exception Handling

- 12.6.1 assert Macro
 - 12.6.2 Failing Fast
 - 12.7 Constructors, Destructors and Exception Handling
 - 12.7.1 Throwing Exceptions from Constructors
 - 12.7.2 Catching Exceptions in Constructors Via Function try Blocks
 - 12.7.3 Exceptions and Destructors; Revisiting noexcept(false)
 - 12.8 Processing new Failures
 - 12.8.1 new Throwing bad_alloc on Failure
 - 12.8.2 new Returning nullptr on Failure
 - 12.8.3 Handling new Failures Using Function set_new_handler
 - 12.9 Standard Library Exception Hierarchy
 - 12.10 C++’s Alternative to the finally Block
 - 12.11 Libraries Often Support Both Exceptions and Error Codes
 - 12.12 Logging
 - 12.13 Looking Ahead to Contracts
 - 12.14 Wrap-Up
-

12.1 Introduction

C++ is used to build real-world, mission-critical and business-critical software. Bjarne Stroustrup (C++’s creator) maintains on his website an extensive list¹ of about 150 applications and systems written partially or entirely in C++. Here are just a few:

1. Bjarne Stroustrup, “C++ Applications,” October 27, 2020. Accessed March 29, 2021.
<https://www.stroustrup.com/applications.html>.

- portions of most major operating systems, like Apple macOS and Microsoft Windows,
- all the compilers we use in this book (GNU g++, Clang and Visual C++),
- Amazon.com,
- aspects of Facebook that require high performance and reliability,

- Bloomberg's real-time financial information systems,
- many of Adobe's authoring, graphics and multimedia applications,
- various database systems, such as MongoDB and MySQL,
- many NASA projects, including aspects of the software in the Mars rovers,
- and much more.

For another extensive list of over 100 applications and systems, see *The Programming Languages Beacon*.²

2. Vincent Lextrait, "The Programming Languages Beacon v16 - March 2016." Accessed March 29, 2021. <https://www.mentofacturing.com/vincent/implementations.html>.

Many of these systems are massive—consider some statistics from the “Codebases: Millions of Lines of Code infographic” (which is not C++ specific):³

3. “Codebases: Millions of Lines of Code infographic.” Accessed March 27, 2021. <https://www.informationisbeautiful.net/visualizations/million-lines-of-code/>. For a spreadsheet of the infographic’s data sources, see <http://bit.ly/CodeBasesInfographicData>.
- A Boeing 787 aircraft’s avionics and online support systems have 6.5 million lines of code, and its flight software is 14 million lines of code.
- The F-35 Fighter Jet has 24 million lines of code.
- The Large Hadron Collider—the world’s largest particle accelerator⁴—in Geneva, Switzerland, has 50 million lines of code.
4. “The Large Hadron Collider.” Accessed March 27, 2021. <https://home.cern/science/accelerators/large-hadron-collider>.
- Facebook has 62 million lines of code.
- An average modern high-end car has 100 million lines of code⁵—across hundreds of processors and controllers.⁶
5. Hans Greimel, “Automakers rush to take back their software codes,” October 12, 2020. Accessed March 27, 2021. <https://www.autonews.com/technology/automakers-rush-take-back-their-software-codes>.
6. Ferenc Valenta’s answer to “How many lines of code are in a car?” January 10, 2019. Accessed March 27, 2021. <https://www.quora.com/How-many-lines-of-code-are-in-a-car/answer/FerencValenta>.

Self-driving cars are expected to require one billion lines of code.⁷ For large and small codebases alike, it's essential to eliminate bugs during development and handle problems that may occur once the software is deployed in real products. That is the focus of this chapter.

7. "Jaguar Land Rover Finds the Teenagers Writing the Code for a Self-driving Future." April 15, 2019. Accessed March 28, 2021. <https://media.jaguarlandrover.com/news/2019/04/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future>.

Exceptions and Exception Handling

SE A As you know, an **exception** is an indication of a problem that occurs during a program's execution. Exceptions may surface through

- explicitly mentioned code in a `try` block,
- calls to other functions (including library calls), and
- operator errors, like `new` failing to acquire additional memory at execution time ([Section 12.8](#)).

Exception handling helps you write robust, **fault-tolerant programs** that catch infrequent problems and:

- deal with them and continue executing,
- perform appropriate cleanup for exceptions that cannot or should not be handled and terminate gracefully, or
- terminate abruptly in the case of unanticipated exceptions—a concept called failing fast, which we discuss in [Section 12.6.2](#).

Exception-Handling, Stack Unwinding and Rethrowing Exceptions

You've seen exceptions get thrown ([Section 6.15](#)) and how to use `try...catch` to handle them ([Section 9.7.12](#)). This chapter reviews these exception-handling concepts in an example that demonstrates the flows of control:

- when a program executes successfully, and
- when an exception occurs.

We discuss use-cases for catching then rethrowing exceptions and show how C++ handles an exception that is not caught in a particular scope. We introduce logging exceptions into a file or database that developers can

review later for debugging purposes.

When to Use Exceptions and Exception Safety Guarantees

Exceptions are not for all types of error handling, so we discuss when and when not to use them. We also introduce the exception safety guarantees you can provide in your code—from none at all to indicating with noexcept that your code does not throw exceptions.

Exceptions in the Context of Constructors and Destructors

We discuss why exceptions are used to indicate errors during construction and why destructors should not throw exceptions. We also demonstrate how to use function `try` blocks to catch exceptions from a constructor's member-initializer list.

Handling Dynamic Memory Allocation Failures

By default, operator `new` throws exceptions when dynamic memory allocation fails. We demonstrate how to catch such `bad_alloc` exceptions. We also show how dynamic memory allocation failures were handled in legacy code before `bad_alloc` was added to C++.

Standard Library Exception Hierarchy and Custom Exception Classes

We introduce the C++ standard library exception-handling class hierarchy. We create a custom exception class that inherits from one of the C++ standard-library exception classes. You'll see why it's important to catch exceptions by reference to enable exception handlers to catch exception types related by inheritance.

Exceptions Are Not Universally Used

Most C++ features have a zero-overhead principle in which you do not pay a price for a given feature unless you use it.⁸ Exceptions violate this principle—programs with exception handling have a larger memory footprint. Some organizations disallow exception handling for this and other reasons. We'll discuss why some libraries provide dual interfaces, enabling developers to choose whether to use versions of functions that throw exceptions or versions that set error codes.

8. Herb Sutter, “De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable,” September 23, 2021. Accessed March 29, 2021. <https://www.youtube.com/watch?>

v=ARYP83yNAWk.

Looking Ahead to Contracts

The chapter concludes with an introduction to contracts. This feature was originally targeted for C++20 but has been delayed to a future version. We'll introduce preconditions, postconditions and assertions, which you'll see are implemented as contracts tested at runtime. If such conditions fail, a contract violation occurs and, by default, the code terminates immediately, enabling you to find errors faster, eliminate them during development and, hopefully, create more robust code for deployment. You'll test the example code using an GCC's experimental contracts implementation on the website godbolt.org.

Exceptions Enable You to Separate Error Handling from Program Logic

  Exception handling provides a standard mechanism for processing errors. This is especially important when working on a large project. As you'll see, using `try...catch` lets you separate the successful path of execution from the error path of execution,^{9,10} making your code easier to read and maintain.¹¹ Also, once an exception occurs, it cannot be ignored—in [Section 12.5](#) you'll see that ignoring an exception can lead to program termination.^{12,13}

9. "Technical Report on C++ Performance," February 15, 2006. Accessed March 26, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf> (page 34).

10. "What does it mean that exceptions separate the good path (or happy path) from the bad path?" Accessed March 24, 2021. <https://isocpp.org/wiki/faq/exceptions#exceptions-separate-good-and-bad-path>.

11. "C++ Exception Handling: Try, Catch, throw Example." Accessed March 24, 2021. <https://www.guru99.com/cpp-exceptions-handling.html>.

12. "Technical Report on C++ Performance," February 15, 2006. Accessed March 26, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf> (page 34).

13. Manoj Piyumal, "Some Useful Facts to Know Before Using C++ Exceptions," December 5, 2017. Accessed March 26, 2021. <https://dzone.com/articles/some-useful-facts-to-know-when-using-c-exceptions>.

Indicating Errors Via Return Values

 Without exception handling, it's common for a function to calculate

and return a value on success or return an error indicator on failure:

- A problem with this architecture is using the return value in a subsequent calculation without first checking whether the value is the error indicator.
- Also, there are cases in which returning error codes is not possible, such as in constructors and overloaded operators.

Exception handling eliminates these problems.

12.2 Exception-Handling Flow of Control; Defining an Exception Class

Let's demonstrate the flow of control:

- when a program executes successfully and
- when an exception occurs.

For demo purposes, Figs. 12.1–12.3 show how to deal with a common arithmetic problem—division-by-zero. In C++, division-by-zero in both integer and floating-point arithmetic is undefined behavior. Each of our preferred compilers issues warnings or errors if they detect integer division-by-zero at compile time. Visual C++ also issues an error if it detects floating-point division-by-zero. At runtime, integer division-by-zero usually causes a program to crash. Some C++ implementations allow floating-point division-by-zero and produce positive or negative infinity—displayed as `inf` or `-inf`, respectively. This is true for each of our preferred compilers.

The example consists of two files:

- `DivideByZeroException.h` ([Fig. 12.1](#)) defines a **custom exception class** (also called a user-defined exception type) representing the type of the problem that might occur in the example, and
- `fig12_02.cpp` ([Fig. 12.3](#)) defines the quotient function and the main function that calls it—we'll use these to explain the exception-handling flow of control.

12.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur

SE  CG  Figure 12.1 defines a custom exception class

`DivideByZeroException` that our program will throw when it detects attempts to divide by zero. We defined this as a derived class of standard library class `runtime_error` (from header `<stdexcept>`). Generally, you should derive your custom exception classes from those in the C++ standard library and name your class so it's clear what problem occurred. The C++ Core Guidelines indicate that such exception classes are less likely to be confused with exceptions thrown by other libraries, like the C++ standard library.¹⁴ We'll say more about the standard exception classes in [Section 12.9](#).

[14. "E.14: Use purpose-designed user-defined types as exceptions \(not built-in types\)." Accessed March 13, 2021.](#)
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-types>.

[Click here to view code image](#)

```
1 // Fig. 12.1: DivideByZeroException.h
2 // Class DivideByZeroException definition.
3 #include <stdexcept> // std::exception header
4
5 // DivideByZeroException objects should be
6 // created by functions upon detecting division-by-
7 class DivideByZeroException : public std::exception
8 public:
9     // constructor specifies default error message
10    DivideByZeroException()
11        : std::runtime_error("attempted to divide by zero")
12 }
```



Fig. 12.1 | Class `DivideByZeroException` definition.

A typical derived class of `runtime_error` defines only a constructor (e.g., lines 10–11) that passes an error-message string to the base-class constructor. You'll soon see that you can process exceptions polymorphically by catching base-class references.

12.2.2 Demonstrating Exception Handling

Figure 12.3 uses exception handling to wrap code that might throw a DivideByZeroException and to handle that exception, should one occur. Our quotient function receives two doubles, divides the first by the second and returns the double result. Function quotient treats all attempts to divide by zero as errors. If it determines the second argument is zero, it **throws a DivideByZeroException** (line 13) to indicate this problem to the caller. **The operand of a throw can be of any copy-constructible type** (not just derived classes of exception). Copy-constructible types are required because the exception mechanism copies the exception into a temporary exception object.¹⁵

15. “throw expression.” Accessed March 13, 2021.
<https://en.cppreference.com/w/cpp/language/throw>.

[Click here to view code image](#)

```
1 // fig12_02.cpp
2 // Example that throws an exception on
3 // an attempt to divide by zero.
4 #include <iostream>
5 #include "DivideByZeroException.h" // Divide
6 using namespace std;
7
8 // perform division and throw DivideByZeroE
9 // divide-by-zero exception occurs
10 double quotient(double numerator, double de
11     // throw DivideByZeroException if trying
12     if (denominator == 0.0) {
13         throw DivideByZeroException{}; // temporary
14     }
15
16     // return division result
17     return numerator / denominator;
18 }
19
20 int main() {
```

```

21     int number1{0}; // user-specified numerator
22     int number2{0}; // user-specified denominator
23
24     cout << "Enter two integers (end-of-file to end): "
25
26     // enable user to enter two integers to end-of-file
27     while (cin >> number1 >> number2) {
28         // try block contains code that might
29         // and code that will not execute if exception occurs
30         try {
31             double result{quotient(number1, number2)};
32             cout << "The quotient is: " << result;
33         }
34         catch (const DivideByZeroException& divideByZeroException) {
35             cout << "Exception occurred: "
36             << divideByZeroException.what();
37         }
38
39         cout << "\nEnter two integers (end-of-file to end): ";
40     }
41
42     cout << '\n';
43 }
```

```

Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 12.2 | Example that throws an exception on an attempt to divide by zero. (Part 2 of 2.)

Assuming that the user does not specify 0 as the denominator for the division, function `quotient` returns the division result. If the user inputs 0 for the denominator, `quotient` throws an exception. In this case, `main` handles the exception, then asks the user to enter two new values before calling `quotient` again. In this way, the program can continue executing even after an improper value is entered, **making the program more robust**. In the sample output, the first two lines show a successful calculation, and the next two show a failure due to an attempt to divide by zero. After we discuss the code, we'll consider the user inputs and flows of control that yield the outputs shown in Fig. 12.3.

12.2.3 Enclosing Code in a `try` Block

 The program prompts the user to enter two integers, then inputs them in the `while` loop's condition (line 27). Line 31 passes the values to `quotient` (lines 10–18), which either divides the integers and returns a result or **throws an exception** if the user attempts to divide by zero. Line 13 in `quotient` is known as the **throw point**. Exception handling is geared to situations where the function that detects an error cannot perform its task.¹⁶

¹⁶. “E.2: Throw an exception to signal that a function can’t perform its assigned task.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-throw>.

The `try` block in lines 30–33 encloses two statements:

- the `quotient` function call (line 31), which can throw an exception, and
- the statement that displays the division result (line 32).

Line 32 executes *only* if `quotient` successfully returns a result.

12.2.4 Defining a `catch` Handler for `DivideByZeroExceptions`

 At least one `catch` handler (lines 34–37) *must* immediately follow each `try` block. **An exception parameter should be declared as a `const reference`¹⁷ to the type the `catch` handler processes**—in this case, a

`DivideByZeroException`). **This has two key benefits:**

17. You could use a non-`const` reference if you need to modify the caught exception object. Typically, you'd do this only with custom exception classes—the standard ones do not provide any member functions that enable you to modify exception objects.

- **Perf** It prevents copying the exception as part of the `catch` operation, which increases performance.
- It enables catching derived-class exceptions.

catching By Reference Avoids Slicing

Err When an exception occurs in a `try` block, **the catch handler that executes is the first one in which the catch's parameter type is either the same as, or a base-class of, the thrown exception's type. Always catch exceptions by reference.** If a base-class catch handler catches a derived-class exception object *by value*, only the base-class portion of the derived-class exception object will be copied into the exception parameter. This is a logic error known as **slicing**, which occurs when you copy or assign a derived-class object into a base-class object.

catch Parameter Name

An exception parameter that includes the *optional* parameter name (as in line 34) enables the `catch` handler to interact with the caught exception—e.g., calling its **what** member function (as in line 36) to get the exception's error message.

Tasks Performed in catch Handlers

Typical tasks performed by `catch` handlers include:

- reporting the error to the user,
- logging errors to a file that developers can study for debugging purposes,
- terminating the program gracefully,
- trying an alternate strategy to accomplish the failed task,
- rethrowing the exception to the current function's caller, or
- throwing a different exception type.

In this example, the `catch` handler simply reports that the user attempted to

divide by zero. Then the program prompts the user to enter two new integer values.

Common Errors When Defining `catch` Handlers

Programmers new to exception handling should be aware of several common coding errors:

- Err  It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.
- Err  Each `catch` handler can have only one parameter—specifying a comma-separated list of exception parameters is a syntax error.
- Err  It's a compilation error to catch the same type in multiple `catch` handlers following a single `try` block.
- Err  It's a logic error to catch a base-class exception before a derived-class exception— compilers typically warn you when this occurs.

12.2.5 Termination Model of Exception Handling

If no exceptions occur in a `try` block, program control continues with the first statement after the last `catch` following that `try` block. If an exception does occur, the `try` block terminates immediately—any local variables defined in that block go out of scope. **This is a strength of exception handling—destructors for the `try` block's local variables are guaranteed to run, minimizing resource leaks.**¹⁸ Next, the program searches for and executes the first matching `catch` handler. If execution reaches that `catch` handler's closing right brace (`}`), the exception is considered handled. Any local variables in the `catch` handler, including the `catch` parameter, go out of scope.

¹⁸. “Technical Report on C++ Performance,” February 15, 2006. Accessed March 26, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf> (page 34).

C++ uses the **termination model of exception handling** in which **program control cannot return to the throw point**. Instead, control resumes with the first statement (line 39) after the `try` block's last `catch` handler.

If an exception occurs in a function and is not caught there, the function terminates immediately. The program attempts to locate an enclosing `try` block in the calling function. This process, called **stack unwinding**, is discussed in [Section 12.5](#).

12.2.6 Flow of Control When the User Enters a Nonzero Denominator

Consider the flow of control in [Fig. 12.3](#) when the user inputs the numerator 100 and the denominator 7. In line 12, `quotient` determines that the denominator is not 0, so line 17 performs the division and returns the result (14.2857) to line 31. Program control continues sequentially from line 31, so line 32 displays the division result, and control reaches the `try` block's ending brace. In this case, the `try` block completed successfully, so program control skips the `catch` handler (lines 34–37) and continues with line 39.

12.2.7 Flow of Control When the User Enters a Zero Denominator

Now consider the flow of control in which the user inputs the numerator 100 and the denominator 0. In line 12, `quotient` determines that the denominator is 0, so line 13 uses keyword `throw` to create and throw a `DivideByZeroException` object. This calls the `DivideByZeroException` constructor to initialize the exception object. Our exception class's constructor does not have parameters. For exception constructors that do, you'd pass the argument(s) in the `throw` statement as you create the object, as in:

[Click here to view code image](#)

```
throw out_of_range{"Index out of range"};
```

 When the exception is thrown, `quotient` exits immediately —*before* it can perform the division. If you explicitly throw an exception from your own code, you generally do so *before* the error has an opportunity to occur. That's not always possible. For example, your function might call another function, which encounters an error and throws an exception.

SE A Because we enclosed the quotient call (line 31) in a `try` block, program control enters the first matching `catch` handler (lines 34–37) that immediately follows the `try` block. In this program, that handler catches `DivideByZeroExceptions`—the type thrown by `quotient`—then prints the error message returned by function `what`. Associating each type of runtime error with an appropriately named exception type improves program clarity.

12.3 Exception Safety Guarantees and `noexcept`

Client-code programmers need to know what to expect when using your code. Is there a potential for exceptions? If so, what will the program’s state be if an exception occurs? When you design your code, consider what **exception safety guarantees**^{19,20} you’ll make:

19. Klaus Iglberger, “Back to Basics: Exceptions,” CPPCON, October 5, 2020, <https://www.youtube.com/watch?v=0ojB8c0xUd8>.

20. “Exceptions.” Accessed March 20, 2021. <https://en.cppreference.com/w/cpp/language/exceptions>.

- **No guarantee**—If an exception occurs, the program may be in an invalid state and resources, like dynamically allocated memory, could be leaked.
- **Basic exception guarantee**—**If an exception occurs, objects’ states remain valid but possibly modified, and no resources are leaked.** Generally, code that can cause exceptions should provide at least a basic exception guarantee.
- **Strong exception guarantee**—**If an exception occurs, objects’ states remain unmodified or, if objects were modified, they’re returned to their original states prior to the operation that caused the exception.** We implemented the copy assignment operator in Chapter 11’s `MyArray` class with a strong exception guarantee via the copy-and-swap idiom. The operator first copies its argument, which could fail to allocate resources. In that case, the assignment fails without modifying the original object; otherwise, the assignment completes successfully.
- **No throw exception guarantee**—**The operation does not throw exceptions.** For example, in Chapter 11, the `MyArray` class’s move constructor, copy constructor and swap functions were declared

`noexcept`. They work with existing resources, rather than allocating new ones, so they cannot fail. Only functions that truly cannot fail should be declared `noexcept`. If an exception occurs in such a function, the program terminates immediately.

12.4 Rethrowing an Exception

A function might use a **resource**, like a file, and might want to **release the resource** (i.e., close the file) **if an exception occurs**. Upon receiving an exception, an exception handler can release the resource then notify its caller that an exception occurred by **rethrowing the exception** with the following statement in a `catch` handler:

```
throw;
```

The next enclosing `try` block detects the rethrown exception, so a `catch` handler listed after that `try` block attempts to handle the exception.

Err Executing the preceding statement outside a `catch` handler terminates the program immediately. If the `catch` handler has a parameter name for the exception it catches, such as `ex`, you can rethrow the exception with

```
throw ex;
```

Perf **Err** However, this unnecessarily copies the original exception object. This also can be a logic error—if the exception handler’s type is a base class of the rethrown exception, **slicing** occurs. For this reason, `throw;` is preferred.

Use Cases for Rethrowing an Exception

There are various use-cases for rethrowing exceptions:

- You might want to log to a file where each exception occurs for future debugging. In this case, you’d catch the exception, log it, then rethrow the exception for further processing in an enclosing scope. We discuss logging in [Section 12.12](#).
- **11** You might want to throw a different exception type that’s more

specific to your library or application. In this scenario, you might wrap the original exception into the new exception by using the C++11 `nested_exception` class.²¹

21. “`std::nested_exception`.” Accessed March 26, 2021. https://en.cppreference.com/w/cpp/error/nested_exception.

- You do partial processing of an exception, such as releasing a resource, then rethrow the exception for further processing in a catch of an enclosing `try` block.
- In some cases, exceptions are implicitly rethrown, such as in function `try` blocks for constructors and destructors, which we discuss in [Section 12.7.2](#).

Demonstrating Rethrowing an Exception

[Figure 12.3](#) demonstrates *rethrowing* an exception. In `main`'s `try` block (lines 25–29), line 27 calls function `throwException` (lines 8–21). The `throwException` function also contains a `try` block (lines 10–13) from which the `throw` statement in line 12 throws an `exception` object. Function `throwException`'s catch handler (lines 14–18) catches this exception, prints an error message (lines 15–16) and rethrows the exception (line 17). This terminates the function and returns control to line 27 in the `try` block in `main`. The `try` block *terminates* (so line 28 does *not* execute), and the catch handler in `main` (lines 30–32) catches this exception and prints an error message (line 31). Since we do not use the exception parameters in this example's catch handlers, we omit the exception parameter names and specify only the type of exception to catch (lines 14 and 30).

[Click here to view code image](#)

```
1 // fig12_03.cpp
2 // Rethrowing an exception.
3 #include <iostream>
4 #include <exception>
5 using namespace std;
6
```

```
7 // throw, catch and rethrow exception
8 void throwException() {
9     // throw exception and catch it immediately
10    try {
11        cout << " Function throwException throws "
12        throw exception{}; // generate exception
13    }
14    catch (const exception&) { // handle exception
15        cout << " Exception handled in function "
16        << "\n Function throwException rethrows "
17        throw; // rethrow exception for further handling
18    }
19
20    cout << "This should not print\n";
21 }
22
23 int main() {
24     // throw exception
25     try {
26         cout << "\nmain invokes function throwException";
27         throwException();
28         cout << "This should not print\n";
29     }
30     catch (const exception&) { // handle exception
31         cout << "\n\nException handled in main";
32     }
33
34     cout << "Program control continues after "
35 }
```



main invokes function throwException
Function throwException throws an exception
Exception handled in function throwException
Function throwException rethrows exception

Exception handled in main

Program control continues after catch in main

Fig. 12.3 | Rethrowing an exception. (Part 2 of 2.)

12.5 Stack Unwinding and Uncaught Exceptions

CG When you do not catch an exception in a particular scope, the function-call stack “unwinds” in an attempt to catch the exception in a catch block of the next outer `try` block—that is, an **enclosing scope**. You can nest `try` blocks, in which case, the next outer `try` block could be in the same function. When `try` blocks are not nested stack unwinding occurs. The function in which the exception was not caught terminates, and its existing local variables go out of scope. During stack unwinding, control returns to the statement that invoked the function. If that statement is in a `try` block, that block terminates, and an attempt is made to catch the exception. If the statement is not in a `try` block or the exception is not caught, stack unwinding continues. **In fact, this mechanism is one reason that you should not wrap a `try...catch` around every function call that might throw an exception²²—sometimes it’s more appropriate to let an earlier function in the call chain deal with the problem.** The program of Fig. 12.4 demonstrates stack unwinding.

22. “E.17: Don’t try to catch every exception in every function.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-not-always>.

[Click here to view code image](#)

```
1 // fig12_04.cpp
2 // Demonstrating stack unwinding.
3 #include <iostream>
4 #include <stdexcept>
5 using namespace std;
6
7 // function3 throws runtime error
8 void function3() {
9     cout << "In function 3\n";
```

```
10     // no try block, stack unwinding occurs,
11     throw runtime_error("runtime_error in fu:
12 }
13
14
15 // function2 invokes function3
16 void function2() {
17     cout << "function3 is called inside func:
18     function3(); // stack unwinding occurs,
19 }
20
21 // function1 invokes function2
22 void function1() {
23     cout << "function2 is called inside func:
24     function2(); // stack unwinding occurs,
25 }
26
27 // demonstrate stack unwinding
28 int main() {
29     // invoke function1
30     try {
31         cout << "function1 is called inside m:
32         function1(); // call function1 which
33     }
34     catch (const runtime_error& error) { // :
35         cout << "Exception occurred: " << err:
36             << "\nException handled in main\n"
37     }
38 }
```



```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Fig. 12.4 | Demonstrating stack unwinding. (Part 2 of 2.)

In main, line 32 in the `try` block calls `function1` (lines 22–25), then `function1` calls `function2` (lines 16–19), which in turn calls `function3` (lines 8–13). Line 12 of `function3` throws a `runtime_error` object—this is the throw point. At this point, control proceeds as follows:

- No `try` block encloses line 12, so stack unwinding begins. `function3` terminates at line 12, returning control to the `function2` statement that called `function3` (i.e., line 18).
- No `try` block encloses line 18, so stack unwinding continues. `function2` terminates at line 18, returning control to the statement in `function1` that invoked `function2` (i.e., line 24).
- Again, no `try` block encloses line 24, so stack unwinding occurs one more time. `function1` terminates at line 24 and returns control to the statement in `main` that invoked `function1` (i.e., line 32).
- The `try` block of lines 30–33 encloses this statement, so the `try` block’s first matching `catch` handler (line 34–37) catches and processes the exception by displaying the exception message.

Uncaught Exceptions

Err~~OK~~ If an exception is not caught during stack unwinding, C++ calls the standard library function `terminate`, which calls `abort` to terminate the program. To demonstrate this, we removed `main`’s `try...catch` in `fig12_04.cpp`, keeping only lines 31–32 from the `try` block in [Fig. 12.4](#). The modified version, `fig12_04modified.cpp`, is in the `fig12_04` example folder. When you execute this modified version of the program and the exception is not caught in `main`, the program terminates. The following shows the output when we executed the program using GNU C++—note that the output mentions `terminate` was called:

[Click here to view code image](#)

function1 is called inside main

```
function2 is called inside function1
function3 is called inside function2
In function 3
terminate called after throwing an instance of 'std::runtime_error'
  what(): runtime_error in function3
Aborted (core dumped)
```

12.6 When to Use Exception Handling

 Exception handling is designed to process infrequent **synchronous errors**, which occur when a statement executes even if your code is correct²³, such as

23. “Modern C++ best practices for exceptions and error handling.” Accessed March 26, 2021. <https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp>.

- accessing a web service that’s temporarily unavailable,
- attempting to read from a file that does not exist,
- attempting to access a file for which you do not have appropriate permissions,
- dynamic memory allocation failures, and more.

 Exception handling is not designed to process errors associated with **asynchronous events**, which occur in parallel with and independent of the program’s flow of control. Examples of these include **I/O completions**, **network message arrivals**, **mouse clicks** and **keystrokes**.

Complex applications usually consist of **predefined software components** (such as standard library classes) and **application-specific components** that use the predefined ones. When a predefined component encounters a problem, it needs to communicate the problem to the application-specific component—the **predefined component cannot know how each application will process a problem**. Sometimes, that problem must be communicated to a function several calls earlier in the function-call chain that led to the exception.

 Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other's error-processing code. It also enables predefined software components (such as standard library classes) to communicate problems to application-specific components. You should incorporate your exception-handling strategy into your system from its inception²⁴—doing so after a system has been implemented can be difficult.

24. “E.1: Develop an error-handling strategy early in a design.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Redesign>.

When Not to Use Exception Handling

The ISO C++ FAQ section on exceptions lists several scenarios in which you should *avoid* using exceptions.²⁵ These include:

25. “What shouldn’t I use exceptions for?” Accessed March 22, 2021. <https://isocpp.org/wiki/faq/exceptions#why-not-exceptions>.

- **cases in which failures are expected**, such as converting incorrectly formatted strings to numeric values where the strings might not have the correct format, out-of-range array indexes and division-by-zero;
- **Perf ⚡ applications that have strict performance requirements where the overhead of throwing exceptions and stack unwinding is unacceptable**, such as the real-time systems used in the United States Joint Strike Fighter plane—for which there is a C++ coding standard;²⁶ and

26. “Joint Strike Fighter Air Vehicle C++ Coding Standards.” December 2005. Accessed March 22, 2021. <https://www.stroustrup.com/JSF-AV-rules.pdf>.

- **frequent errors that should not happen in code**, such as **accessing out-of-range array elements, dereferencing a null pointer and division by zero**. It’s interesting that the C++ standard includes the `out_of_range` exception class, and various C++ standard library classes, such as `array` and `vector`, have member functions that throw `out_of_range` exceptions.

 **Exception handling can increase the executable size of your program²⁷, which may be unacceptable in memory-constrained devices,**

such as embedded systems. According to the ISO C++ FAQ on exception handling, however, “... exception handling is extremely cheap when you don’t throw an exception. It costs nothing on some implementations. All the cost is incurred when you throw an exception: that is, normal code is faster than code using error-return codes and tests. You incur cost only when you have an error.”²⁸ For a detailed discussion of exception-handling performance, see [Section 5.4](#) of the ***Technical Report on C++ Performance***.²⁹

27. Vishal Chovatiya, “C++ Exception Handling Best Practices: 7 Things To Know.” <http://www.vishalchovatiya.com/7-best-practices-for-exception-handling-in-cpp-with-example/>.
28. “Why use exceptions?” Accessed March 26, 2021. <https://isocpp.org/wiki/faq/exceptions#why-exceptions>.
29. “Technical Report on C++ Performance.” February 15, 2006. Accessed March 29, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf>.

 **SE** Functions with common error conditions generally should return `nullptr`, 0 or other appropriate values, such as `bools`, rather than throw exceptions. A program calling such a function can check the return value to determine whether the function call succeeded or failed. Herb Sutter—the ISO C++ language standards committee (WG21) Convener and a Software Architect at Microsoft—indicates, “Programs bugs are not recoverable runtime errors and so should not be reported as exceptions or error codes.” He goes on to say that the process is already underway to migrate the C++ standard libraries away from throwing exceptions for such errors.³⁰ In [Section 12.13](#), Looking Ahead to Contracts, we’ll see that the new contracts capabilities, originally scheduled to be included in C++20 and now deferred until at least C++23, can help reduce the need for exceptions in the standard library.

30. Herb Sutter, “Zero-overhead deterministic exceptions: Throwing values,” August 4, 2019. Accessed March 26, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf>.

 **CG** The C++ Core Guidelines indicate that using lots of `try...catch` statements in your code can be a sign that you’re doing too much low-level resource management.³¹ In this case, they recommend designing your classes to use **RAII (Resource Acquisition Is Initialization)**, which we introduced

in Chapter 11.

31. “E.18: Minimize the use of explicit `try/catch`.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-catch>.

12.6.1 assert Macro

When implementing and debugging programs, it’s sometimes useful to state conditions that should be true at a particular point in a function. These conditions, called **assertions**, help ensure a program’s validity by catching potential bugs and identifying possible logic errors during development. An assertion is a runtime check for a condition that should always be true if your code is correct. If the condition is false, the program terminates immediately, displaying an error message that includes the filename and line number where the problem occurred and the condition that failed. You implement an assertion using the **assert macro**³² from the **<cassert> header**, as in:

32. “assert.” Accessed March 27, 2021. <https://en.cppreference.com/w/cpp/error/assert>.

```
assert (condition);
```

Assertions are primarily a development-time aid³³ that is meant to deal with coding errors that need to be fixed, not something that can be corrected at runtime. For example, you might use an assertion in a function that processes arrays to ensure that the array indexes are greater than 0 and less than the array’s length. Once you’re done debugging, you can disable assertions by adding the following preprocessor directive before the `#include` for the `<cassert>` header

33. “Modern C++ best practices for exceptions and error handling.” August 24, 2020. Accessed March 27, 2021. <https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp>.

```
#define NDEBUG
```

12.6.2 Failing Fast

 The C++ Core Guidelines suggest that “If you can’t throw exceptions, consider failing fast.”³⁴ **Fail-fast**³⁵ is a development style in which, rather than catching an exception, processing it and leaving your

program in a state where it might fail later, the program terminates immediately. This seems counterintuitive. The idea is that failing fast actually helps you build more robust software, because you might be able to find and fix errors sooner during development. Fewer errors are likely to make their way into the final product.³⁶

34. “E.26: If you can’t throw exceptions, consider failing fast.” Accessed March 27, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw-crash>.
35. “Fail-fast.” Accessed March 27, 2021. <https://en.wikipedia.org/wiki/Fail-fast>.
36. Jim Shore, “Fail Fast [Software Debugging].” IEEE Software 21, no. 5 (September/October 2004): 21–25. Edited by Martin Fowler. <https://martinfowler.com/ieeeSoftware/failFast.pdf>.

12.7 Constructors, Destructors and Exception Handling

There are some subtle issues regarding exceptions in the context of constructors and destructors. In this section, you’ll see

- why constructors should throw exceptions when they encounter errors,
- how to catch exceptions that occur in a constructor’s member initializer, and
- why destructors should not throw exceptions.

12.7.1 Throwing Exceptions from Constructors

 First, consider an issue we’ve mentioned but not yet resolved. What happens when an error is detected in a constructor? For example, how should an object’s constructor respond when it receives invalid data? Because **the constructor cannot return a value to indicate an error**, we must somehow indicate that the object has not been constructed properly. One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it’s in an inconsistent state. Another is to set a variable outside the constructor, such as a global error variable, but that’s considered poor software engineering. The preferred alternative is to require the constructor to throw an exception that contains the error

information.

SE A Do not throw exceptions from the constructor of a global object or a static local object. Such exceptions cannot be caught because they're constructed *before* main executes.

SE A A constructor should throw an exception if a problem occurs while initializing an object. Before doing so, the constructor should release any dynamically allocated memory to prevent memory leaks.

12.7.2 Catching Exceptions in Constructors Via Function try Blocks

Recall that base-class initializers and member initializers execute *before* the constructor's body. So, if you want to catch exceptions thrown by those initializers, you cannot simply wrap a constructor's body statements in a **try** block. Instead, you must use a **function try block**, which we demonstrate in Fig. 12.5.

[Click here to view code image](#)

```
1 // fig12_05.cpp
2 // Demonstrating a function try block.
3 #include <iostream>
4 #include <limits>
5 #include <stdexcept>
6 using namespace std;
7
8 // class Integer purposely throws an exception
9 class Integer {
10 public:
11     explicit Integer(int i) : value{i} {
12         cout << "Integer constructor: " << va
13             << "\nPurposely throwing exception"
14         throw runtime_error("Integer constructo
15     }
16 private:
```

```

17     int value{} ;
18 }
19
20 class ResourceManager {
21 public:
22     ResourceManager(int i) try : myInteger(i)
23         cout << "ResourceManager constructor"
24     }
25     catch (runtime_error& ex) {
26         cout << "Exception while constructing"
27             << ex.what() << "\nAutomatically r"
28     }
29 private:
30     Integer myInteger;
31 }
32
33 int main() {
34     try {
35         const ResourceManager resource{7};
36     }
37     catch (const runtime_error& ex) {
38         cout << "Rethrown exception caught in"
39     }
40 }
```

Integer constructor: 7
 Purposely throwing exception from Integer constr
 Exception while constructing ResourceManager: In
 Automatically rethrowing the exception
 Rethrown exception caught in main: Integer const

Fig. 12.5 | Demonstrating a function try block.

Our class `Integer` (lines 9–18) will simulate failing to “acquire a resource.” This class’s constructor purposely throws an exception (line 14) to

help demonstrate a function try block in action. Class ResourceManager (lines 20–31) contains an object of class Integer (line 30), which will be initialized in the ResourceManager constructor's member-initializer list.

In a constructor, you define a function try block by placing the try keyword *after* the constructor's parameter list and *before* the colon (:) that introduces the member initializer list (line 22). The member initializer list is followed by the constructor's body (lines 22– 24). Any exceptions that occur in the member initializer list or in the constructor's body can be handled by catch blocks that follow the constructor's body—in this example, the catch block at lines 25–28. The flow of control in this example is as follows:

- Line 35 in main creates an object of our ResourceManager class, which calls the class's constructor.
- Line 22 in the constructor calls class Integer's constructor (lines 11– 15) to initialize the myInteger object, producing the first two lines of output. Line 14 purposely throws an exception so we can demonstrate the ResourceManager constructor's function try block. This terminates class Integer's constructor and throws the exception back to the base-class initializer in line 22, which is in the ResourceManager constructor's function try block.
- The function try block, which also includes the constructor's body, terminates.
- The ResourceManager constructor's catch handler at lines 25–28 catches the exception and displays the next two lines of output.
- **SE**  The primary purpose of a constructor's function try block is to enable you to do initial exception processing, such as logging the exception or throwing a different exception that's more appropriate for your code. **Your object cannot be fully constructed, so each catch handler that follows a function try block is required to either throw a new exception, or rethrow the existing one—explicitly or implicitly.³⁷** Our catch handler does not explicitly contain a throw statement, so the catch handler implicitly rethrows the exception. This

terminates the ResourceManager constructor and throws the exception back to line 35 in main.

37. “Function-try-block.” Accessed March 22, 2021.
<https://en.cppreference.com/w/cpp/language/function-try-block>.

- Line 35 is in a try block, so that block terminates, and the catch handler in lines 37–39 handles the exception, displaying the last line of the output.

SE  Function try blocks also may be used with other functions, as in:

[Click here to view code image](#)

```
void myFunction() try {
    // do something
}
catch (const ExceptionType& ex) {
    // exception processing
}
```

However, for a regular function, a function try block does not provide any additional benefit over simply placing the entire try...catch sequence in the function’s body, as in:

[Click here to view code image](#)

```
void myFunction() {
    try {
        // do something
    }
    catch (const ExceptionType& ex) {
        // exception processing
    }
}
```

12.7.3 Exceptions and Destructors; Revisiting `noexcept(false)`

11 As of C++11, the compiler implicitly declares all destructors `noexcept`, unless

- you say otherwise by declaring a destructor `noexcept(false)`, or
- a direct or indirect base class's destructor is declared `noexcept(false)`.

SE A If your destructor calls functions that might throw exceptions, you should catch and handle those exceptions, even if that simply means logging the exception and terminating the program in a controlled manner.³⁸ During destruction of a derived class object, if it's possible for a base-class destructor to throw an exception, you can use a function `try` block on your derived-class destructor to ensure that you have the opportunity to catch the exception.

38. “C.36: A **destructor must not fail.**” Accessed March 22, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-fail>.

If an exception occurs during object construction, destructors may be called:

- If an exception is thrown before an object is fully constructed, destructors will be called for any member objects constructed so far.
- If an array of objects has been partially constructed when an exception occurs, only the destructors for the array's constructed objects will be called.

Stack unwinding is guaranteed to have been completed at the point that a catch handler begins executing. **If a destructor invoked as a result of stack unwinding throws an exception, the program terminates.** According to the ISO C++ FAQ³⁹, the choice to terminate the program in this scenario is because C++ does not know whether to:

39. “How can I handle a destructor that fails?” Accessed March 25, 2021. <https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw>.

- continue processing the exception that led to stack unwinding in the first place or
- process the new exception thrown from the destructor.

12.8 Processing `new` Failures

Section 11.4 demonstrated dynamically allocating memory with `new`. Then,

Section 11.5 showed modern C++ memory management using **RAII (Resource Acquisition Is Initialization)**, class template `unique_ptr` and function template `make_unique`, which uses operator `new` “under the hood.” If `new` fails to allocate the requested memory, it throws a `bad_alloc` exception (defined in header `<new>`).

In this section, Figs. 12.6–12.7 present two examples of `new` failing—each attempts to acquire large amounts of dynamically allocated memory:

- The first example demonstrates `new` throwing a `bad_alloc` exception.
- The second uses function `set_new_handler` to specify a function to call when `new` fails. **This technique is mainly used in legacy C++ code that was written before compilers supported throwing a `bad_alloc` exceptions.**

SE  When an exception is thrown from the constructor for an object that’s created in a `new` expression, the dynamically allocated memory for that object is released.

Do Not Throw Exceptions While Holding a Raw Pointer to Dynamically Allocated Memory

CG  Section 11.5 showed that a `unique_ptr` enables you to ensure dynamically allocated memory is properly deallocated regardless of whether the `unique_ptr` goes out of scope due to the normal flow of control or due to an exception. **When managing dynamically allocated memory via old-style raw pointers (as might be the case in legacy C++ code), it’s critical that you do not allow exceptions to occur before you release the memory.**⁴⁰ For example, if a function contained the following series of statements:

40. “E.13: Never throw while being the direct owner of an object.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-never-throw>.

[Click here to view code image](#)

```
int* ptr{new int[100]}; // acquire dynamically al-
processArray(ptr); // assume this function might
delete[] ptr; // return the memory to the system
```

```
// ...
```

```
< >
```

a **memory leak** would occur if `processArray` throws an exception—the code would not reach the `delete[]` statement. To prevent this leak, you'd have to catch the exception and delete the memory before allowing the exception to propagate back to the caller. Instead, you should manage memory with `unique_ptr`, as in:

[Click here to view code image](#)

```
std::unique_ptr<int[]> ptr{std::make_unique<int>()};
processArray(ptr); // possible exception here
// ...
```

```
< >
```

If `processArray` throws an exception, the `unique_ptr` will go out of scope and automatically return the memory to the system.

12.8.1 `new` Throwing `bad_alloc` on Failure

Figure 12.6 demonstrates `new` *implicitly* throwing `bad_alloc` when `new` fails to allocate memory. The `for` statement (lines 15–18) inside the `try` block iterates through the array of `unique_ptr` objects named `items` and allocated to each element an array of 500,000,000 doubles. Our test computer has 32 GB of RAM and eight TB of disk space. We had to allocate enormous numbers of elements to force dynamic memory allocation to fail. You might be able to specify fewer than 500,000,000 on your system. If `new` fails during a call to `make_unique` and throws a `bad_alloc` exception, the loop terminates. The program continues in line 20, where the `catch` handler catches and processes the exception. Lines 21–22 print "Exception occurred:", followed by the message returned from the base-class-exception version of function `what`. Typically, this is an implementation-defined exception-specific message, such as "bad allocation" or "std::bad_alloc". The output shows that the program performed only ten iterations of the loop before `new` failed and threw the `bad_alloc` exception. Your output might differ based on your system's physical memory, the disk space available for virtual memory on your system and the compiler you're using.

[Click here to view code image](#)

```
1 // fig12_06.cpp
2 // Demonstrating standard new throwing bad_
3 // cannot be allocated.
4 #include <array>
5 #include <iostream>
6 #include <memory>
7 #include <new> // bad_alloc class is defined
8 using namespace std;
9
10 int main() {
11     array<unique_ptr<double[]>, 1000> items{
12
13     // aim each unique_ptr at a big block of
14     try {
15         for (int i{0}; auto& item : items) {
16             item = make_unique<double[]>(500'0
17             cout << "items[" << i++ << "] point"
18         }
19     }
20     catch (const bad_alloc& memoryAllocation
21         cerr << "Exception occurred: "
22             << memoryAllocationException.what(
23     }
24 }
```

< >

```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
```

```
items[8] points to 500,000,000 doubles
items[9] points to 500,000,000 doubles
Exception occurred: bad allocation
```

Fig. 12.6 | new throwing bad_alloc on failure.

12.8.2 new Returning nullptr on Failure

The C++ standard specifies that programmers can use an older version of new that returns nullptr upon failure. For this purpose, header <new> defines object **nothrow** (of type **nothrow_t**), which is used as follows:

[Click here to view code image](#)

```
unique_ptr<double[]> ptr{new(nothrow) double[500]}
```

Sec  Unfortunately, we cannot use make_unique in this scenario because it uses the default version of new. To make programs more robust, use the version of new that throws bad_alloc exceptions on failure.

12.8.3 Handling new Failures Using Function **set_new_handler**

In **legacy C++ code**, you might encounter another feature for handling new failures—the **set_new_handler** function (header <new>). This function takes as its argument either:

- a pointer to a function that takes no arguments and returns void, or
- a lambda that takes no arguments and does not return a value.

The function or lambda is called if new fails. This provides a uniform approach to handling all new failures, regardless of where a failure occurs in the program. **Once set_new_handler registers a new handler in the program, operator new does not throw bad_alloc on failure. Instead, it delegates the error handling to the new-handler function.**

If new allocates memory successfully, it returns a pointer to that memory. If new fails to allocate memory and set_new_handler did not register a new-handler function, new throws a bad_alloc exception. If new fails to

allocate memory and a new-handler function has been registered, the new-handler function is called.

The new-handler function should perform one of the following tasks:

1. Make more memory available by deleting other dynamically allocated memory or telling the user to close other applications, then try allocating memory again.
2. Throw an exception of type `bad_alloc`.
3. Call function `abort` or `exit` (both found in header `<cstdlib>`) to terminate the program. **The `abort` function terminates a program immediately, whereas `exit` executes destructors for global objects and local static objects before terminating the program. Non-static local objects are not destructed when either of these functions is called.**

Figure 12.7 demonstrates `set_new_handler`. Function `customNewHandler` (lines 10–13) prints an error message (line 11), then calls `exit` (line 12) to terminate the program. The constant `EXIT_FAILURE` is defined in the header `<cstdlib>`, which is included in many C++ standard library headers. The output shows that the loop iterated nine times before `new` failed and invoked function `customNewHandler`. Your output might differ based on your compiler, and the physical memory and disk space available for virtual memory on your system.

[Click here to view code image](#)

```
1 // fig12_07.cpp
2 // Demonstrating set_new_handler.
3 #include <array>
4 #include <iostream>
5 #include <memory>
6 #include <new> // set_new_handler is defined
7 using namespace std;
8
9 // handle memory allocation failure
10 void customNewHandler() {
11     cerr << "customNewHandler was called\n";
```

```
12     exit(EXIT_FAILURE);
13 }
14
15 int main() {
16     array<unique_ptr<double[]>, 1000> items{
17
18     // specify that customNewHandler should
19     // memory allocation failure
20     set_new_handler(customNewHandler);
21
22     // aim each unique_ptr at a big block of
23     for (int i{0}; auto& item : items) {
24         item = make_unique<double[]>(500'000'
25         cout << "items[" << i++ << "] points"
26     }
27 }
```

< >

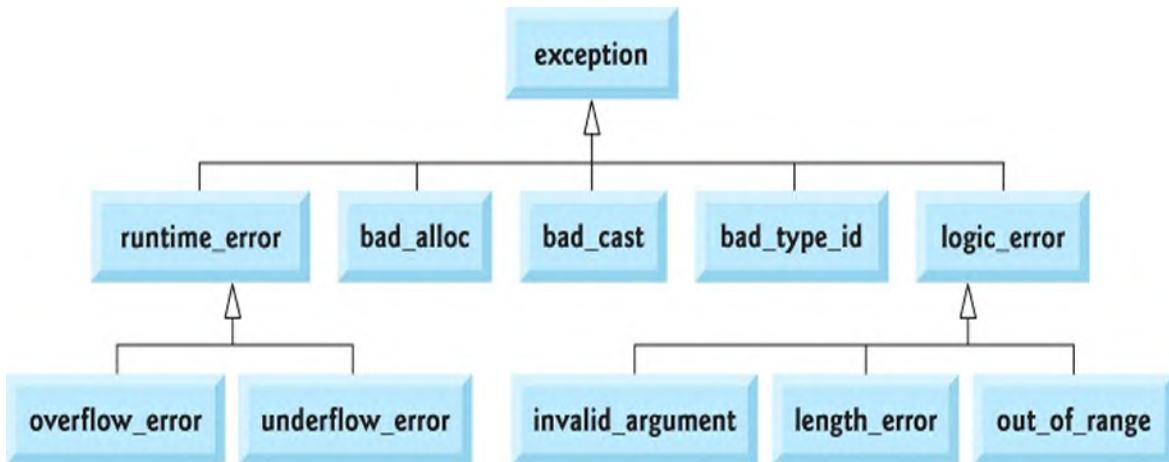
```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
items[8] points to 500,000,000 doubles
customNewHandler was called
```

Fig. 12.7 | `set_new_handler` specifying the function to call when `new` fails. (Part 2 of 2.)

12.9 Standard Library Exception Hierarchy

Exceptions fall nicely into several categories. The C++ standard library includes a hierarchy of exception classes, some of which are shown in the

following diagram:



For a list of the 28 exception types in the C++ standard library, see:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/error/exception>

You can build programs that can throw

- standard exceptions,
- exceptions derived from the standard exceptions,
- your own exceptions not derived from the standard exceptions or
- instances of non-class types, like fundamental-type values and pointers.

SE A CG The `exception` class hierarchy is a good starting point for creating custom exception types. In fact, rather than throwing exceptions of the types shown in the preceding diagram, **the C++ Core Guidelines recommend creating derived-class exception types that are specific to your application**. Such custom types can convey more meaning than the generically named exception types, like `runtime_error`.

Base Class `exception`

CG The standard library exception hierarchy is headed by base-class `exception` (defined in header `<exception>`). This class contains virtual function what that derived classes can override to issue an appropriate error message. If a catch handler specifies a reference to a base-class exception type, it can catch objects of all exception classes

derived publicly from that base class.⁴¹

41. “E.15: Catch exceptions from a hierarchy by reference.” Accessed March 13, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-ref>.

Derived Classes of exception

Immediate derived classes of exception include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes. Also derived from `exception` are the exceptions thrown by C++ operators:

- `bad_alloc` is thrown by `new` (Section 12.8),
- `bad_cast` is thrown by `dynamic_cast` (Section 19.7) and
- `bad_typeid` is thrown by `typeid` (Section 19.7).

Derived Classes of runtime_error

Class `runtime_error`, which we used briefly in Section 12.5, is the base class of several other standard exception classes that indicate execution-time errors:

- Class `overflow_error` describes an **arithmetic overflow error** (i.e., the result is larger than the largest number that can be stored in a given numeric type).
- Class `underflow_error` describes an **arithmetic underflow error** (i.e., the result is smaller than the smallest number that can be stored in a given numeric type).

Derived Classes of logic_error

Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic:

- We used class `invalid_argument` in `set` functions (starting in Chapter 9) to indicate when an attempt was made to set an invalid value. Proper coding can, of course, prevent invalid arguments from reaching a function.
- Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.

- Class **out_of_range** indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Catching Exception Types Related By Inheritance

SE  Using inheritance with exceptions enables an exception handler to catch related errors with concise notation:

- **Err**  One approach is to `catch` each type of reference to a derived-class exception object individually. This is error-prone—you could forget to test explicitly for one or more of the derived-class types.
- A more concise approach is to `catch` references to base-class exception objects.

It's a logic error if you place a base-class `catch` handler before one that catches one of that base class's derived types. The base-class `catch` matches all objects of classes derived from that base class, so the derived-class `catch` will never execute.⁴²

⁴². “E.31: Properly order your `catch`-clauses.” Accessed March 13, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re_ca

Catching All Exceptions

Err  **SE**  C++ exceptions need not derive from class **exception**, so catching type **exception** is not guaranteed to catch all exceptions a program could encounter. You can use `catch(...)` to catch all exception types thrown in a `try` block. There are weaknesses to this approach:

- The type of the caught exception is unknown.
- Also, without a named parameter, you cannot refer to the exception object inside the exception handler.

SE  The `catch(...)` handler is primarily used to perform recovery that does not depend on the exception type, such as releasing common resources. **The exception can be rethrown to alert enclosing catch handlers.**

12.10 C++’s Alternative to the `finally` Block

In several programming languages that were created after C++—such as Java, C# and Python—the `try` statement has an optional `finally` block that is guaranteed to execute, regardless of whether the corresponding `try` block completes successfully or terminates due to an exception. The `finally` block is placed after a `try` block’s last exception handler or immediately after the `try` block if there are no exception handlers (which is allowed in those languages but not in C++). This makes `finally` blocks in those other languages a good mechanism for guaranteeing resource deallocation to prevent resource leaks.

As a programmer in another language, you might wonder why C++’s `try` statement has not added a `finally` block. In C++, we do not need `finally` due to **RAII (Resource Acquisition Is Initialization)**, smart pointers and destructors. If you design a class to use RAII, objects of your class will acquire their resources during object construction and deallocate them during object destruction.

Over the years, similar capabilities have been added to Java, C# and Python as well:

- Java has `try-with-resources` statements.
- C# has `using` statements.
- Python has `with` statements.

As program control enters these statements, each creates objects that acquire resources, which you can then use in the statements’ bodies. When these statements terminate—successfully or due to an exception—they deallocate the resources automatically.

12.11 Libraries Often Support Both Exceptions and Error Codes

Exceptions are not universally used. A 2018 ISO worldwide C++ developer survey showed that exceptions were partially or fully banned in 52% of projects.⁴³ For example,

⁴³. “C++ Developer Survey ‘Lite’: 2018-02.” Accessed March 25, 2021.

<https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf>.

- the **Google C++ Style Guide**⁴⁴ indicates that they do not use exceptions, and

44. “Google C++ Style Guide.” Accessed March 22, 2021. <https://google.github.io/styleguide/cppguide.html>.

- the **Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards**⁴⁵ explicitly forbids using try, catch and throw.

45. “Joint Strike Fighter Air Vehicle C++ Coding Standards.” December 2005. Accessed March 22, 2021. <https://www.stroustrup.com/JSF-AV-rules.pdf>.

When organizations prohibit exceptions, they’re also prohibiting use of libraries that contain functions with the potential to throw exceptions—such as the C++ standard library.

There are various disadvantages to not allowing exceptions in projects. Some problems one developer encountered in a project that banned exceptions included:⁴⁶

46. Lucian Radu Teodorescu’s answer to “Why do some people recommend not using exception handling in C++? Is this just a "culture" in C++ community, or do some real reasons exist behind this?” August 2, 2015. Accessed March 25, 2021. <https://www.quora.com/Why-do-some-people-recommend-not-using-exception-handling-in-C--Is-this-just-a-culture-in-C--community-or-do-some-real-reasons-exist-behind-this/answer/Lucian-Radu-Teodorescu>.

- interoperability issues with class libraries that use exceptions,
- the amount of code required to deal with error conditions,
- problems with errors during object construction,
- problems with overloaded assignment operators,
- difficulties with error-handling flows of control,
- cluttering of the code by intermixing of logic and error handling,
- issues with resource allocation and deallocation, and
- efficiency of the code.

To give programmers the flexibility of choosing whether to use exceptions, some libraries support dual interfaces with two versions of each function:

- one that throws an exception when it encounters a problem, and

- one that sets or returns an error indicator when it encounters a problem.

As an example of this, consider the functions in C++17's `<filesystem>` library⁴⁷. These functions enable you to manipulate files and folders from C++ applications. In this library, each function has two versions—one that throws a `filesystem_error` exception and one that sets a value in its `error_code` argument that you pass to the function by reference.

⁴⁷. <https://en.cppreference.com/w/cpp/filesystem>.

12.12 Logging

One common task when handling exceptions is to log where they occurred into a human-readable text file that developers can analyze later for debugging purposes. Logging can be used during development time to help locate and fix problems. It also can be used once an application ships—if an application crashes, it might write a log file that the user can then send to the developer.

Logging is not built into the C++ standard library, though you could create your own logging mechanisms using C++'s file-processing capabilities. However, there are many open source C++ logging libraries:

- Boost.Log
—https://www.boost.org/doc/libs/1_75_0/libs/log/c
- Easylogging++
—<https://github.com/amrayn/easyloggingpp>.
- Google Logging Library (glog)
—<https://github.com/google/glog>.
- Loguru—<https://github.com/emilk/loguru>.
- Plog—<https://github.com/SergiusTheBest/plog>.
- spdlog—<https://github.com/gabime/spdlog>.

Some of these are header-only libraries that you can simply include in your projects. Others require installation procedures.

12.13 Looking Ahead to Contracts⁴⁸

⁴⁸. Contracts are not yet a standard C++ feature. The syntax we show here could change before

contracts eventually become part of the language.

To strengthen your program's error-handling architecture, you can specify the expected states before and after a function's execution with preconditions and postconditions, respectively. We'll define and show code examples of each, and explain what happens when they are violated.

- A **precondition**⁴⁹ must be true when a function is invoked. Preconditions describe constraints on function parameters and any other expectations the function has just before it begins executing. **If the preconditions are not met, then the function's behavior is undefined**—it may throw an exception, proceed with an illegal value or attempt to recover from the error. If code with undefined behavior is allowed to proceed, the results could be unpredictable and not portable across platforms. Each function can have multiple preconditions.

49. "Precondition." Accessed March 27, 2021.
<https://en.wikipedia.org/wiki/Precondition>.

- A **postcondition**⁵⁰ is true after the function successfully returns. Postconditions describe constraints on the return value or side effects the function may have. When defining a function, you should document all postconditions so that others know what to expect when they call your function. You also should ensure that your function honors its postconditions if its preconditions are met. Each function can have multiple postconditions.

50. "Postcondition." Accessed March 27, 2021.
<https://en.wikipedia.org/wiki/Postcondition>.

Precondition and Postcondition Violations

Today, precondition and postcondition violations often are dealt with by throwing exceptions. Consider array and vector function `at`, which receives an index into the container. For a precondition, function `at` requires that its index argument be greater than or equal to 0 and less than the container's size. If the precondition is met, `at`'s postcondition states that the function will return the item at that index; otherwise, `at` throws an `out_of_range` exception. As a client of an array or vector, we trust that function `at` satisfies its postcondition, provided that we meet the precondition.

Preconditions and postconditions are assertions, so you can implement them with the `assert` preprocessor macro (Section 12.6.1) as program control enters or exits a function. Preprocessor macros are generally deprecated in C++. Until contracts become part of C++, you'll continue using the `assert` macro or custom C++ code to express preconditions, assertions and postconditions.

Invariants

An **invariant** is a condition that should always be true in your code—that is, a condition that never changes. **Class invariants** must be true for each object of a class. They generally are tied to an object's lifecycle. Class invariants remain true from the time an object is constructed until it's destructed. For example:

- Section 9.6's `Account` class requires that its `m_balance` data member always be nonnegative.
- Section 9.7's `Time` class requires that its `m_hour` data member always have a value in the range 0 through 23, and its `m_minute` and `m_second` members always have values in the range 0 through 59.

These invariants ensure that objects of these classes always maintain valid state information throughout their lifetimes. Functions also may contain invariants. For example, in a function that searches for a specified value in a `vector<int>`, the invariant is that if the value is in the vector, the value's index must be greater than or equal to 0 and less than the vector's size.

Design By Contract

Design by contract (DbC)^{51,52,53} is a **software-design approach** created by **Bertrand Meyer in the 1980s and used in the design of his Eiffel programming language**. Using this approach:

51. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

52. Bertrand Meyer. In *Touch of Class: Learning to Program Well with Objects and Contracts*, xvii. Springer Berlin AN, 2016.

53. “Design by contract.” Accessed March 28, 2010.
https://en.wikipedia.org/wiki/Design_by_contract.

- a function expects client code to meet the function's precondition(s),

- if the preconditions are true, the function guarantees that its postcondition(s) will be true, and
- any invariants are maintained.

23 A proposal to add support for contract-based programming (commonly referred to as “contracts”) to the C++ standard was first proposed in 2012 and later rejected.⁵⁴ Another proposal was eventually accepted for inclusion in C++20, but removed⁵⁵ late in C++20’s development cycle due to “lingering design disagreements and concerns.”⁵⁶ So, contracts have been pushed to at least C++23.

54. Nathan Meyers, “What Happened to C++20 Contracts?” August 5, 2019. Accessed March 28, 2021.

https://www.reddit.com/r/cpp/comments/cmk7ek/what_happened_to_c20_

55. Nathan Meyers, “What Happened to C++20 Contracts?” August 5, 2019. Accessed March 28, 2021.

https://www.reddit.com/r/cpp/comments/cmk7ek/what_happened_to_c20_

56. Herb Sutter, “Trip report: Summer ISO C++ standards meeting (Cologne),” July 2019. Accessed March 28, 2021. <https://herbsutter.com/2019/07/20/trip-report-summer-iso-c-standards-meeting-cologne/>.

Gradually Moving to Contracts in the C++ Standard Library

Herb Sutter says, “in Java and .NET some 90% of all exceptions are thrown for precondition violations.” He also says, “The programming world now broadly recognizes that programming bugs (e.g., out-of-bounds access, null dereference, and in general all pre/ post/assert-condition violations) cause a corrupted state that cannot be recovered from programmatically, and so they should never be reported to the calling code as exceptions or error codes that code could somehow handle.”⁵⁷

57. Herb Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil),” July 2018. Accessed March 22, 2021. <https://herbsutter.com/2018/07/>.

SE  A key idea behind incorporating contracts is that many errors we currently deal with via exceptions can be located via preconditions and postconditions then eliminated by fixing the code.⁵⁸

58. Glennen Carnie, “Contract killing (in Modern C++),” September 18, 2019. Accessed March 28, 2021. <https://blog.feabhas.com/2019/09/contract-killing-in-modern-c/>.

Perf A goal of contracts is to make most functions `noexcept`,⁵⁹ which will enable the compiler to perform additional optimizations. Sutter says, that “Gradually switching precondition violations from exceptions to contracts promises to eventually remove a majority of all exceptions thrown by the standard library.”^{60,61}

59. Herb Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil),” July 2018. Accessed March 22, 2021. <https://herbsutter.com/2018/07/>.

60. Herb Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil),” July 2018. Accessed March 22, 2021. <https://herbsutter.com/2018/07/>.

61. To get a sense of the number of exceptions thrown by C++ and its libraries, we searched for the word “throws” in the final draft of the C++ standard document located at <https://isocpp.org/files/papers/N4860.pdf>. The document is over 1800 pages—450+ pages cover the language, 1000+ cover the standard library and the rest are appendices, bibliography, cross references and indexes. “Throws” appears 422 times—there were 80 occurrences of “throws nothing,” 13 occurrences of “throws nothing unless...” (indicating an exception that is thrown as a result of stack unwinding) and 329 occurrences of functions that throw exceptions. Many of these 329 cases are examples of where contracts will help eliminate the need to throw exceptions.

Contracts Attributes

The contracts proposal⁶² introduces three attributes of the form

62. G. Dos Reis, J.D. Garcia, J. Lakos, A. Meredith, N. Meyers, B. Stroustrup, “Support for contract based programming in C++,” June, 8, 2018. Accessed March 28, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html>.

[[contractAttribute optionalLevel optionalIdentifier: condition]]

that you can use to specify preconditions, postconditions and assertions for your functions. The *optionalIdentifier* is one of the function’s local variables for use in postconditions, as you’ll see in Fig. 12.9. The contract attributes are:

- **expects**—for specifying a function’s preconditions that are checked before the function’s body begins executing,
- **ensures**—for specifying a function’s postconditions that are checked just before the function returns, and
- **assert**—for specifying assertions that are checked as they’re encountered throughout a function’s execution.

If you specify multiple preconditions and postconditions, they’re checked in

their order of declaration.

Contracts Levels

There are three contract levels:

- **default** specifies a contract that has little runtime overhead compared to the function's typical execution time. If a level is not specified, the compiler assumes default.
- **Perf  SE  audit** specifies a contract that has **significant runtime overhead** compared to the function's typical execution time. Such contracts are intended primarily for use during program development.
- **SE  axiom** specifies a contract that is meant to be enforced by static code checkers, rather than at runtime.

Perf  Using these levels will enable you to select which contracts are enforced at runtime, thus controlling the performance overhead. You can choose—presumably via compiler flags—whether to turn contracts off entirely, perform the low-overhead default contracts or perform the high-overhead audit contracts.

Specifying Preconditions, Postconditions and Assertions

Precondition contracts (`expects`) and postcondition contracts (`ensures`) are specified in a function's prototype—they are listed after the function's signature and before the semicolon. For example, a function that calculates the real (not imaginary) square root of a `double` value expects its argument to be greater than or equal to zero. You can specify this with an `expects` precondition contract:

[Click here to view code image](#)

```
double squareRoot(double value)
  [[expects: value >= 0.0]];
```

If the function definition also serves as the function prototype, the precondition and postcondition contracts are listed between the function's signature and its opening left brace.

Assertions are specified as statements in the function body. For instance, to check whether an integer `exam grade` is in the range 0 through 100, you'd write:

[Click here to view code image](#)

```
[[assert: grade >= 0 && grade <= 100]];
```

Note that the `assert` in the preceding statement is not an `assert` macro.

Early-Access Implementation

There is an early-access contracts implementation in GNU C++⁶³ that you can test through the **Compiler Explorer website**⁶⁴ (<https://godbolt.org>), which supports many compiler versions across various programming languages. You can choose “**x86-64 gcc (contracts)**” as your compiler and compile contracts-based code using the compiler options described at:

63. There is also an early-access Clang contracts implementation at <https://github.com/arcosuc3m/clang-contracts>, but you need to build and install it yourself.
64. Copyright © 2012–2019, Compiler Explorer Authors. All rights reserved. Compiler Explorer is by Matt Godbolt. <https://xania.org/MattGodbolt>.

[Click here to view code image](#)

<https://gitlab.com/lock3/gcc-new/-/wikis/contract>



We provide a `godbolt.org` URL where you can try each of our examples using the early-access implementation. The GNU C++ early-access contracts implementation uses different keywords for preconditions and postconditions:

- **pre** rather than `excepts`, and
- **post** rather than `ensures`.

Example: Division By Zero

Figure 12.8 reimplements our quotient function from Fig. 12.3. Here, we specify in the quotient function's prototype (lines 6–7) a default level precondition contract indicating that the denominator must not be 0.0.

The `default` keyword also can be specified explicitly, as in:

[Click here to view code image](#)

```
[[pre default: denominator != 0.0]]
```

This example can be found at:

[Click here to view code image](#)

<https://godbolt.org/z/fWxTE5ov9>

As you type code into the Compiler Explorer editor, or when you load an existing example, Compiler Explorer automatically compiles and runs it or displays any compilation errors.

[Click here to view code image](#)

```
1 // fig12_08.cpp
2 // quotient function with a contract precondition
3 #include <iostream>
4 using namespace std;
5
6 double quotient(double numerator, double denominator,
7     [[pre: denominator != 0.0]]);
8
9 int main() {
10    cout << "quotient(100, 7): " << quotient(100, 7)
11    << "\nquotient(100, 0): " << quotient(100, 0)
12 }
13
14 // perform division
15 double quotient(double numerator, double denominator,
16     return numerator / denominator;
17 }
```



Fig. 12.8 | quotient function with a contract precondition.

We preset the compiler options for this example to

```
-std=c++2a -fcontracts
```

20 which compiles the code with C++20 and experimental contracts support —c++2a was the early-access GNU C++ compiler notation for C++20. Compiler Explorer generally shows at least two tabs—a code editor and a compiler. Our examples also show the output tab so you can see the executed program’s results—this is also where compilation errors would be displayed. The compiler tab has two drop-down lists at the top. One lets you select the compiler. The other lets you view and edit the compiler options, or you can click the down arrow to select common compiler options.

The quotient call at line 10 satisfies the precondition and executes successfully, producing:

```
quotient(100, 7): 14.2857
```

The quotient call at line 11, however, causes a **contract violation**. So, the **default violation handler (handle_contractViolation)** is implicitly called, displays the following error message then terminates the program:

[Click here to view code image](#)

```
default std::handle_contractViolation called:  
./example.cpp 7 quotient denominator != 0.0 default  
< >
```

Changing the compilation options to

[Click here to view code image](#)

```
-std=c++2a -fcontracts -fcontract-build-level=off
```

disables contract checking and allows line 11 to execute, producing the output:

```
quotient(100, 0): inf
```

Recall that division by zero is undefined behavior in C++. However, many compilers, including the three key compilers we use throughout this book, return positive or negative infinity (`inf` or `-inf`) in this case. This is the behavior specified by the **IEEE 754 standard for floating-point arithmetic**, which is widely supported by modern programming languages.

Contract Continuation Mode

The default **continuation mode** for contract violations is to terminate the program immediately. To allow a program to continue executing, you can add the compiler option

[Click here to view code image](#)

```
-fcontract-continuation-mode=on
```

Example: Binary Search

Consider Fig. 12.9, which defines a `binarySearch` function template with a precondition and a postcondition. To save space, we show only the prototype here. The complete example, which you can find at:

[Click here to view code image](#)

<https://godbolt.org/z/K1qxf8MYY>

contains the `binarySearch` function template's definition.

[Click here to view code image](#)

```
1 // fig12_09.cpp
2 // binarySearch function with a precondition and a postcondition
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6 using namespace std;
7
8 template <typename T>
9 int binarySearch(const vector<T>& items, const T& target) {
10     [[pre audit: is_sorted(begin(items), end(items))]];
11     [[post loc: loc == -1 || (loc >= 0 && loc < items.size())]];
12 }
```

```

12
13     int main() {
14         // sorted vector v1 satisfies binarySearch
15         vector v1{10, 20, 30, 40, 50, 60, 70, 80}
16         int result1 = binarySearch(v1, 70);
17         cout << "70 was " << (result1 != -1 ? ""
18
19         // unsorted vector v2 violates binarySearch
20         vector v2{60, 70, 80, 90, 10, 20, 30, 40}
21         int result2 = binarySearch(v2, 60);
22         cout << "60 was " << (result2 != -1 ? ""
23     }

```



Fig. 12.9 | `binarySearch` function with a precondition and a postcondition.

The `binarySearch` function template performs a binary search on a vector. This algorithm requires the vector to be in sorted order; otherwise, the result could be incorrect. So we declared the precondition

[Click here to view code image](#)

```
[[pre audit: is_sorted(begin(items), end(items))]]
```



which calls the C++ standard library function `is_sorted` (from header `<algorithm>`⁶⁵) to check whether the vector is sorted. This is potentially an expensive operation. A binary search of a billion element sorted vector requires only 30 comparisons. However, determining whether the vector is sorted requires 999,999,999 comparisons. And sorting one billion elements efficiently with an $O(n \log_2 n)$ sort algorithm could require about 30 billion operations. A developer might want to enable this test during development and disable it in production code. For this reason, we specified the precondition contract level audit.

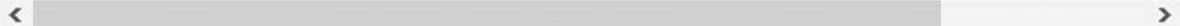
⁶⁵. There are over 200 functions in the C++ standard library's `<algorithm>` header. We survey many of these in [Chapter 14](#).

The `binarySearch` function template's prototype also specifies the

postcondition:

[Click here to view code image](#)

```
[[post loc: loc == -1 || (loc >= 0 && loc < items
```



This demonstrates an optional identifier (`loc`) from the function body that can be used in the postcondition contract. If our `binarySearch` function template does not find the search key, it will return the `loc` value `-1`. Otherwise, `loc` will be an index that's greater than or equal to `0` and less than the `vector`'s size.

We preset the compiler options for this example to

```
-std=c++2a -fcontracts
```

20 which, again, compiles the code with C++20 and default level contracts support, so the **line 10 audit level precondition contract is ignored**. In `main`, we created two `vectors` of integers containing the same values—`v1` is sorted (line 15), and `v2` is unsorted (line 20). With the initial compiler settings, the precondition is not tested, so the `binarySearch` calls in lines 16 and 21 complete, and the program displays the output:

```
70 was found in v1
60 was not found in v2
```

Because the **audit level** precondition was ignored, our program has a logic error. The second line of output shows that `60` was not found in `v2`. Again, the algorithm expects `v2` to be sorted, so it failed to find `60`, even though it's in `v2`. Changing the **contract build level** to `audit` in the compilation options, as in:

[Click here to view code image](#)

```
-std=c++2a -fcontracts -fcontract-build-level=aud
```



enables audit level contract checking. When the program runs with audit contracts enabled, the `vector v1` in line 16's `binarySearch` call satisfies the precondition in line 10 and, as before, line 17 outputs:

```
70 was found in v1
```

However, vector v2 in line 21's binarySearch call causes a **contract violation**, resulting in the error message:

[Click here to view code image](#)

```
default std::handle_contractViolation called:  
./example.cpp 10 binarySearch<int> is_sorted(begin  
default 0
```

The postcondition in our example is always true, because we coded the binarySearch function correctly. To show a postcondition contract violation, we

- modified binarySearch to incorrectly return -2 when the key is not found, and
- changed the binarySearch call in line 16 to search for a key that's not in v1.

This forces binarySearch to return -2, violating the postcondition. You can view the modified code at:

[Click here to view code image](#)

<https://godbolt.org/z/EbbYGET7n>

In this case, the postcondition causes a contract violation and produces the output:

[Click here to view code image](#)

```
default std::handle_contractViolation called:  
./example.cpp 11 binarySearch<int> loc == -1 || ([  
loc < items.size()) default default 0
```

Custom Contract Violation Handler

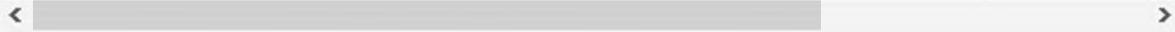
The examples so far used the default contract violation handler. When a contract violation occurs, a `contractViolation` object is created containing the following information:

- the line number of the violation—returned by member function `lineNumber`,
- the source-code file name—returned by member function `fileName`,
- the function name in which the violation occurred—returned by member function `functionName`,
- a description of the condition that was violated—returned by member function `comment`, and
- the contract level—returned by member function `assertionLevel`.

The `contractViolation` is passed to the **violation handler** function of the form:

[Click here to view code image](#)

```
void handle_contractViolation(const contract_vio·
{
    // handler code
}
```



12.14 Wrap-Up

In this chapter, we discussed that C++ is used to build real-world, mission-critical and business-critical software. The systems it's used for are often massive. You learned that it's essential to eliminate bugs during development and decide how to handle problems once the software is in production.

We overviewed the ways that exceptions may surface in your code. We discussed how exception handling helps you write robust, fault-tolerant programs that catch infrequent problems and continue executing, perform appropriate cleanup and terminate gracefully, or terminate abruptly in the case of unanticipated exceptions.

We reviewed exception-handling concepts in an example that demonstrated the flows of control when a program executes successfully, and when an

exception occurs. We discussed use-cases for catching then rethrowing exceptions. We showed how stack unwinding enables other functions to handle an exception that is not caught in a particular scope.

You learned when to use exceptions. We also introduced the exception guarantees you can provide in your code—no guarantee, a basic exception guarantee, a strong exception guarantee and no-throw exception guarantee. We discussed why exceptions are used to indicate errors during construction and why destructors should not throw exceptions. We also showed how to use function `try` blocks to catch exceptions from a constructor's member-initializer list or from base-class destructors when a derived-class object is destroyed.

You saw that operator `new` throws `bad_alloc` exceptions when dynamic memory allocation fails. We also showed how dynamic memory allocation failures were handled in legacy C++ code with `set_new_handler`.

We introduced the C++ standard library exception class hierarchy and created a custom exception class that inherited from a C++ standard library exception class. You learned why it's important to catch exceptions by reference to enable exception handlers to catch exception types related by inheritance and to avoid slicing. We also introduced logging exceptions into a file that developers can analyze later for debugging purposes.

We discussed why some organizations disallow exception handling. You also saw that some libraries provide dual interfaces, so developers can choose whether to use versions of functions that throw exceptions or versions that set error codes.

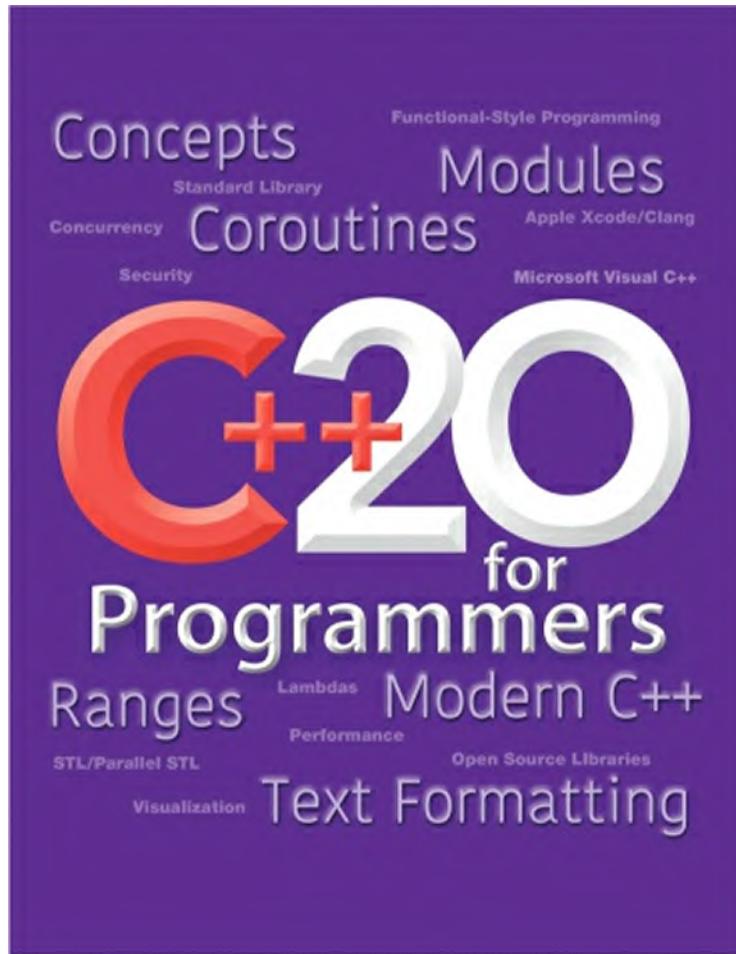
The chapter concluded with an introduction to the contracts feature, originally adopted for C++20 but delayed to a future C++ version. We showed how to use contracts to test preconditions, postconditions and assertions at runtime. You learned that these test conditions which should always be true in correct code. You saw that if such conditions are false, contract violations occur and, by default, the code terminates immediately. This enables you to find errors faster, eliminate them during development and, hopefully, create more robust code.

[Chapter 7](#) introduced the `array` and `vector` standard library classes. In [Chapter 13](#), you'll learn about many additional C++ standard library containers as well as iterators, which are used by standard library algorithms

to walk through containers and manipulate their elements.

Part 4: Standard Library Containers, Iterators and Algorithms

Chapter 13. Standard Library Containers and Iterators



Objectives

In this chapter, you'll:

- Be introduced to the standard library containers, iterators and algorithms—a world of reusable software capabilities.
- Understand how containers relate to C++20 ranges.
- Use I/O stream iterators to read values from the standard input stream and write values to the standard output stream.

- Use iterators to access container elements.
 - Use the `vector`, `list` and `deque` sequence containers.
 - Use `ostream_iterator` with the `std::copy` and `std::ranges::copy` algorithms to output container elements.
 - Use the `set`, `multiset`, `map` and `multimap` ordered associative containers.
 - Understand the differences between the ordered and unordered associative containers.
 - Use the `stack`, `queue` and `priority_queue` container adaptors.
 - Understand how to use the `bitset` “near container” to manipulate a collection of bit flags.
-

Outline

13.1 Introduction

13.2 Introduction to Containers

13.2.1 Common Nested Types in Sequence and Associative Containers

13.2.2 Common Container Member and Non-Member Functions

13.2.3 Requirements for Container Elements

13.3 Working with Iterators

13.3.1 Using `istream_iterator` for Input and `ostream_iterator` for Output

13.3.2 Iterator Categories

13.3.3 Container Support for Iterators

13.3.4 Predefined Iterator Type Names

13.4 A Brief Introduction to Algorithms

13.5 Sequence Containers

13.6 `vector` Sequence Container

13.6.1 Using vectors and Iterators

13.6.2 `vector` Element-Manipulation Functions

- 13.7** `list` Sequence Container
 - 13.8** `deque` Sequence Container
 - 13.9** Associative Containers
 - 13.9.1** `multiset` Associative Container
 - 13.9.2** `set` Associative Container
 - 13.9.3** `multimap` Associative Container
 - 13.9.4** `map` Associative Container
 - 13.10** Container Adaptors
 - 13.7.1** `stack` Adaptor
 - 13.7.2** `queue` Adaptor
 - 13.7.3** `priority_queue` Adaptor
 - 13.11** `bitset` Near Container
 - 13.12** Optional: A Brief Intro to Big *O*
 - 13.13** Optional: A Brief Intro to Hash Tables
 - 13.14** Wrap-Up
-

13.1 Introduction

The standard library defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. We began introducing templates in [Chapters 5–6](#) and use them extensively here and in [Chapters 14](#) and [15](#). Historically, the features presented in this chapter were referred to as the **Standard Template Library** or **STL**.¹ In the C++ standard document, they are simply referred to as part of the C++ standard library.

¹ The STL was developed by Alexander Stepanov and Meng Lee at Hewlett Packard and is based on their generic programming research, with significant contributions from David Musser. Stepanov first proposed the STL for inclusion in C++ at the November 1993 ANSI/ISO C++ standardization committee meeting, and it was approved for inclusion in July 1994 (https://en.wikipedia.org/wiki/History_of_the_Standard_Template_Lib accessed April 15, 2021).

Containers, Iterators and Algorithms

This chapter introduces three key components of the standard library —**containers** (templatized data structures), iterators and algorithms. We'll introduce **containers**, **container adaptors** and **near containers**.

Common Member Functions Among Containers

Each container has associated member functions—a subset of these is defined in all containers. We illustrate most of this common functionality in our examples of **array** (introduced in [Chapter 6](#)), **vector** (also introduced in [Chapter 6](#) and covered in more depth here), **list** ([Section 13.7](#)) and **deque** (pronounced “deck”; [Section 13.8](#)).

Iterators

Iterators, which have properties similar to those of **pointers**, are used to manipulate container elements. **Built-in arrays** also can be manipulated by standard library algorithms, using pointers as iterators. We’ll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with standard library algorithms—in some cases, reducing many lines of code to a single statement.

Algorithms

Standard library **algorithms** (which we’ll cover in depth in [Chapter 14](#)) are function templates that perform common data manipulations, such as **searching, sorting, copying, transforming** and **comparing elements or entire containers**. The standard library provides scores of algorithms. There are:

- 20** • 90 in the `<algorithms>` header’s `std` namespace—82 also are overloaded in the `std::ranges` namespace for use with C++20 ranges,
- 11 in the `<numeric>` header’s `std` namespace,

- 20** • 14 in the `<memory>` header’s `std` namespace—all 14 also are overloaded in the `std::ranges` namespace for use with C++20 ranges,
- 2 in the `<cstdlib>` header.

Many were added in C++11 and C++20, and a few in C++17. For the complete list of algorithms with links to their descriptions, visit:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/algorithm>

Most algorithms use iterators to access container elements. Each algorithm has minimum requirements for the kinds of iterators that can be used with it. We'll see that containers support specific kinds of iterators, some more powerful than others. The iterators a container supports determine whether the container can be used with a specific algorithm. Iterators encapsulate the mechanisms used to traverse containers and access their elements. This encapsulation enables many of the algorithms to be applied to various containers independently of the underlying container implementation. This also enables you to create new algorithms that can process the elements of multiple container types.

20 C++20 Ranges

[Chapter 6](#) introduced C++20's new **ranges** and **views**. You saw that a **range** is a collection of elements you can iterate over. So, **arrays** and **vectors** are ranges. You also used **views** to specify **pipelines** of operations that manipulate ranges of elements. Any container that has iterators representing its beginning and end can be treated as a C++20 range. In this chapter, we'll use the new C++20 standard library algorithm `std::ranges::copy` and the older C++ standard library algorithm `std::copy` to demonstrate how ranges simplify your code. In [Chapter 14](#), we'll use many more C++20 algorithms from the `std::ranges` namespace to demonstrate additional **ranges** and **views** features.

Custom Templatized Data Structures

Some popular data structures include linked lists, queues, stacks and binary trees:

- **Linked lists** are collections of data items logically “lined up in a row”— insertions and removals are made anywhere in a linked list.
- **Stacks** are important in compilers and operating systems: Insertions and removals are made **only** at one end of a stack—its **top**.
- **Queues** represent waiting lines; insertions are made at the back (also referred to as the **tail**) of a queue and removals are made from the front (also referred to as the **head**) of a queue.
- **Binary trees** are nonlinear, hierarchical data structures that facilitate **searching** and **sorting** data, **duplicate elimination** and **compiling**

expressions into machine code.

Each of these data structures has many other interesting applications. We can carefully weave linked objects together with pointers. Pointer-based code is complex and can be error prone—the slightest omissions or oversights can lead to serious **memory-access violations** and **memory leaks** with no forewarning from the compiler. If many programmers on a large project implement custom containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug.

 Avoid reinventing the wheel. When possible, program with the C++ standard library’s preexisting containers, iterators and algorithms.² The prepackaged standard library container classes provide the data structures you need for most applications. Using the standard library’s proven containers, iterators and algorithms helps you reduce testing and debugging time. The containers, iterators and algorithms were conceived and designed for performance and flexibility.

2. “SL.1: Use libraries wherever possible.” Accessed April 16, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-lib>.

13.2 Introduction to Containers³

3. This section is intended as an introduction to a reference-oriented chapter. You may want to read it quickly and refer back to it as necessary when reading the chapter’s live-code examples.

The standard library containers are divided into four major categories:

- **sequence containers**,
- **ordered associative containers**,
- **unordered associative containers** and
- **container adaptors**.

We briefly summarize the containers here and show live-code examples of many of them in the following sections.

Sequence Containers

The **sequence containers** represent linear data structures with all of their elements conceptually “lined up in a row,” such as **arrays**, **vectors** and

linked lists. The five **sequence containers** are:

- **array** ([Chapter 6](#))—Fixed size. Direct access to any element.
 - **deque** ([Section 13.8](#))—Rapid insertions and deletions at front or back. Direct access to any element.
- 11** • **forward_list**—Singly linked list, rapid insertion and deletion anywhere.
- **list** ([Section 13.7](#))—Doubly linked list, rapid insertion and deletion anywhere.
 - **vector** ([Section 13.6](#))—Rapid insertions and deletions at back. Direct access to any element.

Class `string` supports the same functionality as a **sequence container**, but stores only character data.

Associative Containers

Associative containers are nonlinear data structures that typically can locate elements stored in the containers quickly. Such containers can store sets of values or **key–value pairs** in which each key has an associated value—for example, a program might associate employee IDs with `Employee` objects. As you’ll see, some associative containers allow multiple values for each key. The keys in associative containers are **immutable**—they cannot be modified unless you first remove them from the container. The four **ordered associative containers** are:

- **set** ([Section 13.9.2](#))—Rapid lookup, no duplicates allowed.
- **multiset** ([Section 13.9.1](#))—Rapid lookup, duplicates allowed.
- **map** ([Section 13.9.4](#))—One-to-one mapping, no duplicates allowed, rapid key-based lookup.
- **multimap** ([Section 13.9.3](#))—One-to-many mapping, duplicates allowed, rapid key-based lookup.

The four **unordered associative containers** are:

- **unordered_set**—Rapid lookup, no duplicates allowed.
- **unordered_multiset**—Rapid lookup, duplicates allowed.

- **`unordered_map`**—One-to-one mapping, no duplicates allowed, rapid key-based lookup.
- **`unordered_multimap`**—One-to-many mapping, duplicates allowed, rapid key-based lookup.

Container Adaptors

Stacks and queues are typically constrained versions of sequence containers. For this reason, the standard library implements **`stack`**, **`queue`** and **`priority_queue`** as **container adaptors** that enable a program to view a sequence container in a constrained manner. The three **container adaptors** are:

- **`stack`**—Last-in, first-out (LIFO) data structure.
- **`queue`**—First-in, first-out (FIFO) data structure.
- **`priority_queue`**—Highest-priority element is always the first element out.

Near Containers

There are other container types that are considered **near containers**—built-in arrays (Chapter 7), bitsets (Section 13.11) for maintaining sets of flag values and valarrays for performing high-speed mathematical vector operations⁴—not to be confused with the vector container. These types are considered **near containers** because they exhibit some, but not all, capabilities of the **sequence** and **associative containers**.

4. For overviews of valarray and its mathematical capabilities, see its documentation at <https://en.cppreference.com/w/cpp/numeric/valarray> and check out the article “std:: valarray class in C++” at <https://www.geeksforgeeks.org/std-valarray-class-c/>.

13.2.1 Common Nested Types in Sequence and Associative Containers

The following table shows the common **nested types** defined inside each sequence container and associative container class definition. These are used in template-based variables declarations, parameters to functions and return values from functions, as you’ll see in this chapter and Chapter 14. For example, the `value_type` in each container always represents the

container's element type.

Nested type	Description
allocator_type	The type of the object used to allocate the container's memory—not included in the <code>array</code> container. Containers that use allocators each provide a default allocator, which is sufficient for most programmers. Providing custom allocators is beyond this book's scope.
value_type	The type of the container's elements.
reference	The type used to declare a reference to a container element.
const_reference	The type used to declare a <code>const</code> reference to a container element.
pointer	The type used to declare a pointer to a container element.
const_pointer	The type used to declare a pointer to a <code>const</code> container element.
iterator	An iterator that points to an element of the container's element type.
const_iterator	An iterator that points to an element of the container's element type. Used only to <code>read</code> elements and to perform <code>const</code> operations.
reverse_iterator	A reverse iterator that points to an element of the container's element type. Iterates through a container <code>back-to-front</code> . This type is not provided by class <code>forward_list</code> .
const_reverse_iterator	A reverse iterator that points to an element of the container's element type and can be used only to <code>read</code> elements and to perform <code>const</code> operations. Used to iterate through a container in reverse. This type is not provided by class <code>forward_list</code> .
difference_type	A type representing the number of elements between two iterators that refer to elements of the same container. A value of this type is returned by an iterators's overloaded <code>-</code> operator, which is not defined for <code>list</code> iterators and associative container iterators).
size_type	The type used to count items in a container and index through a sequence container (cannot index through a <code>list</code>).

13.2.2 Common Container Member and Non-Member

Functions

Most containers provide similar functionality. Many operations apply to all containers, and other operations apply to subsets of similar containers. The following tables describe the functions that are commonly available in most, but not all, standard library containers. Before using any container, you should study its capabilities. For a complete list of all the container member functions and which containers support them, see the **member function table** at [cppreference.com](https://en.cppreference.com).⁵ Several of the member functions that we do not list in this section are covered throughout the chapter as we present various sequence containers, associative containers and container adaptors.

5. “Container library—Member functions table.” Accessed April 16, 2021.
<https://en.cppreference.com/w/cpp/container>.

Container Special Member Functions

The following table describes the container special member functions provided by each container. In addition, each container class typically provides many overloaded constructors for initializing containers and container adaptors in various ways. For example, each sequence and associative container can be initialized from an `initializer_list`.

Container special member function	Description
default constructor	A constructor that initializes an empty container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
copy operator=	Copies the elements of one container into another.
move constructor	Moves the contents of an existing container into a new one of the same type. The old container no longer contains the data. This avoids the overhead of copying each element of the existing container.
move operator=	Moves the elements of one container into another of the same type. The old container no longer contains the data. This avoids the overhead of copying each element of the existing container.
destructor	Destructor function for cleanup after a container is no longer needed.

Non-Member Relational and Equality Operators

20 The following table shows the relational and equality operators supported by most containers. In C++17 and earlier, the `<`, `<=`, `>`, `>=`, `==` and `!=` operators are overloaded as non-member functions. In C++20, the sequence containers and associative containers replace these overloaded operator functions with the new **three-way comparison operator** `<=>`. As we showed in [Section 11.7](#), the C++ compiler can use `<=>` to implement each relational and equality comparison by rewriting it in terms of `<=>`. For example, the compiler rewrites

`x == y`

as

`(x <=> y) == 0`

The unordered associative containers do not support `<`, `<=`, `>` and `>=`. The relational and equality operators are not supported for `priority_queues`.

Non-member operator functions	Description
<code>operator<</code>	Returns <code>true</code> if the contents of the first container are less than the second; otherwise, returns <code>false</code> .
<code>operator<=</code>	Returns <code>true</code> if the contents of the first container are less than or equal to the second; otherwise, returns <code>false</code> .
<code>operator></code>	Returns <code>true</code> if the contents of the first container are greater than the second; otherwise, returns <code>false</code> .
<code>operator>=</code>	Returns <code>true</code> if the contents of the first container are greater than or equal to the second; otherwise, returns <code>false</code> .
<code>operator==</code>	Returns <code>true</code> if the contents of the first container are equal to the contents of the second; otherwise, returns <code>false</code> .
<code>operator!=</code>	Returns <code>true</code> if the contents of the first container are not equal to the contents of the second; otherwise, returns <code>false</code> .

Member Functions That Return Iterators

The following table shows container member functions that return iterators. The **`forward_list`** container does have the member functions `rbegin`, `rend`, `crbegin` and `crend`.

Container member function	Description
<code>begin</code>	Returns an <code>iterator</code> or a <code>const_iterator</code> —depending on whether the container is <code>const</code> —referring to the container's first element.
<code>end</code>	Returns an <code>iterator</code> or a <code>const_iterator</code> —depending on whether the container is <code>const</code> —referring to the next position after the end of the container.
<code>cbegin</code> (C++11)	Returns a <code>const_iterator</code> referring to the container's first element.
<code>cend</code> (C++11)	Returns a <code>const_iterator</code> referring to the next position after the end of the container.
<code>rbegin</code>	Returns a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> —depending on whether the container is <code>const</code> —referring to the container's last element.
<code>rend</code>	Returns a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> —depending on whether the container is <code>const</code> —referring to the position before the container's first element.
<code>crbegin</code> (C++11)	Returns a <code>const_reverse_iterator</code> referring to the container's last element.
<code>crend</code> (C++11)	Returns a <code>const_reverse_iterator</code> referring to the position before the container's first element.

Other Member Functions

The following table lists various additional container member functions. If a function is not supported for all containers, we specify the containers that do or do not support it.

Container member function	Description
20 <code>clear</code>	Removes all elements from the container. This is not supported for arrays .
11 <code>contains</code> (C++20)	Returns <code>true</code> if the specified value is present in the container; otherwise, returns <code>false</code> . This is supported only for associative containers.
11 <code>empty</code>	Returns <code>true</code> if there are no elements in the container; otherwise, returns <code>false</code> .
11 <code>emplace</code> (C++11)	Inserts an item into the container using move semantics. If the item is a temporary object constructed in <code>emplace</code> 's argument list, <code>emplace</code> tries to construct it in place where it will reside in the container; otherwise, it constructs the object then moves it into place. This is not supported for arrays .
17 <code>erase</code>	Removes one or more elements from the container.
17 <code>extract</code> (C++17)	The associative containers are linked data structures of nodes containing values. Associative container member function <code>extract</code> removes a node from the container and returns an object of that container's <code>node_type</code> .
11 <code>insert</code>	Inserts an item in the container. This is overloaded to support <code>copy</code> and <code>move</code> semantics.
11 <code>max_size</code>	Returns the maximum number of elements for a container.
11 <code>size</code>	Returns the number of elements currently in the container.
11 <code>swap</code>	Swaps the elements of two containers. As of C++11, there is a non-member-function version of <code>swap</code> that swaps the contents of its two arguments (which must be of the same container type) using <code>move</code> operations rather than <code>copy</code> operations.

13.2.3 Requirements for Container Elements

Before using a standard library container, it's important to ensure that the type of objects being stored in the container supports a minimum set of functionality. For example:

- If inserting an item into a container requires a **copy** of the object, the object type should provide a **copy constructor** and **copy assignment operator**.
- If inserting an item into a container requires **moving** the object, the

object type should provide a **move constructor** and **move assignment operator**—[Chapter 11](#) discussed **move semantics**.

- The **ordered associative containers** and many algorithms require elements to be compared—for this reason, the object type should support comparisons.

As you review the documentation for each container, whether in the C++ standard document itself or on sites like `cppreference.com`, you'll see various **named requirements**, such as **CopyConstructible**, **MoveAssignable** or **EqualityComparable**. These help you determine whether your types are compatible with various C++ standard library containers and algorithms. You can view a list of named requirements and their descriptions at:

[Click here to view code image](#)

`https://en.cppreference.com/w/cpp/named_req`

20 In C++20, many named requirements are formalized as **concepts**, which we'll say more about in [Chapters 14](#) and [15](#).

13.3 Working with Iterators

Iterators have many similarities to pointers. Iterators point to elements in sequence containers and associative containers. Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented for each type of container. Certain iterator operations are uniform across containers. For example, the **dereferencing operator (*)** dereferences an iterator so that you can use the element to which it points. The **++ operation on an iterator** moves it to the container's next element.

Sequence containers and associative containers provide member functions `begin` and `end`. Function **begin** returns an iterator pointing to the first element of the container. Function **end** returns an iterator pointing to the **first element past the end of the container** (one past the end)—a non-existent element that's frequently used to determine when the end of a container is reached. This is typically used in an equality or inequality comparison to determine whether a “moving iterator” reached the end of the

container.

An object of a container's **iterator** type refers to a container element that can be modified, and an object of a container's **const_iterator** type refers to a container element that cannot be modified.

13.3.1 Using **istream_iterator** for Input and **ostream_iterator** for Output

We use iterators with **sequences** (also called **ranges**). These can be in containers, or they can be **input sequences** or **output sequences**. Figure 13.1 demonstrates input from the standard input (a sequence of data for program input), using an **istream_iterator**, and output to the standard output (a sequence of data for program output), using an **ostream_iterator**. The program inputs two integers from the user and displays their sum. As you'll see later in this chapter, **istream_iterators** and **ostream_iterators** can be used with the standard library algorithms to create powerful statements. For example, subsequent examples will use an **ostream_iterator** with the **copy** algorithm to copy a container's elements to the standard output stream with a single statement.

[Click here to view code image](#)

```
1 // fig13_01.cpp
2 // Demonstrating input and output with iterator
3 #include <iostream>
4 #include <iterator> // ostream_iterator and
5
6 int main() {
7     std::cout << "Enter two integers: ";
8
9     // create istream_iterator for reading integers
10    std::istream_iterator<int> inputInt{std::cin};
11
12    const int number1{*inputInt}; // read in
13    ++inputInt; // move iterator to next input
14    const int number2{*inputInt}; // read in
```

```
15
16     // create ostream_iterator for writing i:
17     std::ostream_iterator<int> outputInt{std
18
19     std::cout << "The sum is: ";
20     *outputInt = number1 + number2; // output
21     std::cout << "\n";
22 }
```

< >

Enter two integers: 12 25

The sum is: 37

Fig. 13.1 | Demonstrating input and output with iterators. (Part 2 of 2.)

istream_iterator

Line 10 creates an `istream_iterator` that's capable of **extracting** (inputting) `int` values from the standard input object `cin`. Line 12 **dereferences** iterator `inputInt` to read the first integer from `cin` and assigns that integer to `number1`. The dereferencing operator `*` applied to iterator `inputInt` gets the value from the stream associated with `inputInt`; this is similar to dereferencing a pointer. Line 13 positions iterator `inputInt` to the next value in the input stream. Line 14 inputs the next integer from `inputInt` and assigns it to `number2`.

ostream_iterator

Perf  Line 17 creates an `ostream_iterator` that's capable of **inserting** (outputting) `int` values in the standard output object `cout`. Line 20 outputs an integer to `cout` by assigning to `*outputInt` the sum of `number1` and `number2`. Notice that we use the dereferenced `outputInt` iterator as an *lvalue* in the assignment statement. If you want to output another value using `outputInt`, the iterator first must be incremented with `++`. Either the prefix or postfix increment can be used—we use the prefix form for performance reasons because it does not create a temporary object.

Err The `*` (dereferencing) operator when applied to a `const` iterator returns a reference to `const` for the container element, disallowing the use of non-`const` member functions.

13.3.2 Iterator Categories

The following table describes the iterator categories. Each provides a specific set of functionality.

Iterator Category	Description
input	Used to read an element from a container. An input iterator can move only forward one element at a time from the container's beginning to its end. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only forward one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice. For the subsequent iterator types in this table, if they refer to non-constant data, the iterators can be used to write to the container as well.
forward	Has the capabilities of an input iterator and retains its position in the container. Such iterators can be used to pass through a sequence more than once for multipass algorithms .
bidirectional	Has the capabilities of a forward iterator and adds the ability to move backward from the container's end toward its beginning. Bidirectional iterators support multipass algorithms.
random access	Has the capabilities of a bidirectional iterator and adds the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements. These can also be compared with relational operators.
17 contiguous	A random-access iterator that requires the elements it operates on to be stored in contiguous memory locations. This category was formalized in C++17.

Throughout this chapter, we discuss which iterator category each container supports. In [Chapter 14](#), you'll see that each algorithm specifies minimum

iterator requirements. Only containers that support those requirements can be used with that algorithm.

13.3.3 Container Support for Iterators

The iterator category that each container supports determines whether that container can be used with specific algorithms. **Containers that support random-access iterators can be used with all standard library algorithms** —with the exception that if an algorithm requires changes to a container's size, the algorithm can't be used on built-in arrays or **array** objects. Pointers into built-in arrays can be used in place of iterators with most algorithms. The following table shows the iterator category of each container. Sequence containers, associative containers, strings and built-in arrays are all traversable with iterators.

Container	Iterator type	Container	Iterator type
Sequence containers		Unordered associative containers	
<code>vector</code>	random access	<code>unordered_set</code>	bidirectional
<code>array</code>	random access	<code>unordered_multiset</code>	bidirectional
<code>deque</code>	random access	<code>unordered_map</code>	bidirectional
<code>list</code>	bidirectional	<code>unordered_multimap</code>	bidirectional
<code>forward_list</code>	forward		
Ordered associative containers		Container adaptors	
<code>set</code>	bidirectional	<code>stack</code>	none
<code>multiset</code>	bidirectional	<code>queue</code>	none
<code>map</code>	bidirectional	<code>priority_queue</code>	none
<code>multimap</code>	bidirectional		

13.3.4 Predefined Iterator Type Names

The following table shows the predefined iterator type names found in the standard library container class definitions. Not every iterator type name is defined for every container. **The `const` iterators are for traversing `const` containers or non-`const` containers that should not be modified.**

Reverse iterators traverse containers in the reverse direction.

Predefined iterator type name	Direction of ++	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

 **Operations performed on a `const_iterator` return references to `const` to prevent modifying elements of the container.** Using `const_iterators` where appropriate is another example of the **principle of least privilege**.

13.3.5 Iterator Operations

The following table shows operations that can be performed on each iterator type. In addition to the operators shown for all iterators, iterators must provide **default constructors**, **copy constructors** and **copy assignment operators**.⁶ A **forward iterator** supports ++ and all of the **input and output iterator** capabilities. A **bidirectional** iterator supports -- and all the capabilities of **forward** iterators. **Random-access** iterators and **contiguous** iterators support all of the operations shown in the table. For input iterators and output iterators, it's not possible to save the iterator, then use the saved value later.

6. You'll see in [Chapter 14](#) that iterators can be wrapped as `move_iterators` to enable move semantics for the referenced elements.

Iterator operation	Description
All iterators	
<code>++p</code>	Preincrement an iterator.
<code>p++</code>	Postincrement an iterator.
<code>p = p1</code>	Assign one iterator to another.
Input iterators	
<code>*p</code>	Dereference an iterator as an <i>rvalue</i> .
<code>p->m</code>	Use the iterator to read the element <code>m</code> .
<code>p == p1</code>	Compare iterators for equality.
<code>p != p1</code>	Compare iterators for inequality.
Output iterators	
<code>*p</code>	Dereference an iterator as an <i>lvalue</i> .
<code>p = p1</code>	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
<code>--p</code>	Predecrement an iterator.
<code>p--</code>	Postdecrement an iterator.
Random-access iterators	
<code>p += i</code>	Increment the iterator <code>p</code> by <code>i</code> positions.
<code>p -= i</code>	Decrement the iterator <code>p</code> by <code>i</code> positions.
<code>p + i or i + p</code>	Expression value is an iterator positioned at <code>p</code> incremented by <code>i</code> positions.
<code>p - i</code>	Expression value is an iterator positioned at <code>p</code> decremented by <code>i</code> positions.
<code>p - p1</code>	Expression value is an integer representing the distance (that is, number of elements) between two elements in the same container.
<code>p[i]</code>	Return a reference to the element offset from <code>p</code> by <code>i</code> positions
<code>p < p1</code>	Return <code>true</code> if iterator <code>p</code> is less than iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p <= p1</code>	Return <code>true</code> if iterator <code>p</code> is less than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p > p1</code>	Return <code>true</code> if iterator <code>p</code> is greater than iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p >= p1</code>	Return <code>true</code> if iterator <code>p</code> is greater than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .

13.4 A Brief Introduction to Algorithms

20 The standard library provides scores of **algorithms** you can use to manipulate a wide variety of containers. **Inserting, deleting, searching, sorting** and others are appropriate for some or all of the sequence and associative containers. **The algorithms operate on container elements only indirectly through iterators.** Many algorithms operate on sequences of elements defined by iterators pointing to the **first element** of the sequence and to **one element past the last element** and many support C++20 ranges. It's also possible to create your own new algorithms that operate in a similar fashion so they can be used with the standard library containers and iterators. In this chapter, we'll use the **copy algorithm** in many examples to copy a container's contents to the standard output. We discuss many standard library algorithms in [Chapter 14](#).

13.5 Sequence Containers

The C++ Standard Template Library provides five **sequence containers** —**array**, **vector**, **deque**, **list** and **forward_list**. The **array**, **vector** and **deque** containers are typically based on built-in arrays. The **list** and **forward_list** containers implement linked-list data structures. We've already discussed and used **array** extensively in [Chapter 6](#), so we do not cover it again here. We've also already introduced **vector** in [Chapter 6](#) —and we discuss it in more detail here.

Perf Performance and Choosing the Appropriate Container

CG  Section 13.2.2 presented the operations common to most of the standard library containers. Beyond these operations, each container typically provides a variety of other capabilities. Many of these are common to several containers, but they're not always equally efficient for each container. **vector is satisfactory for most applications.**⁷

7. “SL.con.2: Prefer using STL `vector` by default unless you have a reason to use a different container.” Accessed April 16, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-vector>.

Perf  Insertion at the back of a **vector** is efficient. The **vector** simply grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a **vector**—the entire portion of the **vector** after the insertion (or deletion) point must be moved, because **vector** elements occupy contiguous cells in memory.

Perf  Applications that require frequent insertions and deletions at both ends of a container normally use a **deque** rather than a **vector**. Although we can insert and delete elements at the front and back of both a **vector** and a **deque**, class **deque** is **O(1)** for insertions and deletions at the front,⁸ making it more efficient than **vector**, which is **O(n)** for those operations.⁹ If you're not familiar with the **Big O notation** used here, see [Section 13.12](#) for a friendly introduction.

8. “`std::deque`.” Accessed April 11, 2021. <https://en.cppreference.com/w/cpp/container/deque>.

Perf  Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a **list**, due to its efficient implementation of insertion and deletion anywhere in the data structure.

13.6 **vector** Sequence Container

The **vector** container, which we introduced in [Section 6.15](#), provides a dynamic data structure with contiguous memory locations. This enables efficient, direct access to any element of a **vector** via the subscript operator `[]`, exactly as with a built-in array. A **vector** is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted, and when the number of elements may need to grow. When a **vector**'s memory is exhausted, the **vector**

- **allocates** a larger built-in array,
- **copies** or **moves** (depending on what the element type supports) the original elements into the new built-in array, and
- **deallocates** the old built-in array.

9. “`std::vector`.” Accessed April 11, 2021. <https://en.cppreference.com/w/cpp/container/vector>.

Perf **Perf** **vectors** provide rapid indexed access with the overloaded subscript operator [] because they're stored in contiguous memory like a built-in array or an **array** object. **Choose the vector container for the best random-access performance in a container that can grow.**

13.6.1 Using vectors and Iterators

Figure 13.2 illustrates several **vector** member functions. Many of these are available in every **sequence container** and **associative container**. You must include header <vector> to use a **vector**. As we add items to a **vector<int>** in this example, we call our showResult function (lines 9–12) to display the added value as well as the **vector**'s

- **size**—the number of elements the **vector** currently contains, and
- **capacity**—the number of elements that the **vector** can store before it needs to **dynamically resize itself** to accommodate more elements.

[Click here to view code image](#)

[Click here to view code image](#)

```
1 // fig13_02.cpp
2 // Standard library vector class template.
3 #include <fmt/format.h> // C++20: This will
4 #include <iostream>
5 #include <ranges>
6 #include <vector> // vector class-template
7
8 // display value appended to vector and upd
9 void showResult(int value, size_t size, siz
10     std::cout << fmt::format("appended: {};\n"
11                     "value, size, capacity);
12 }
13
```



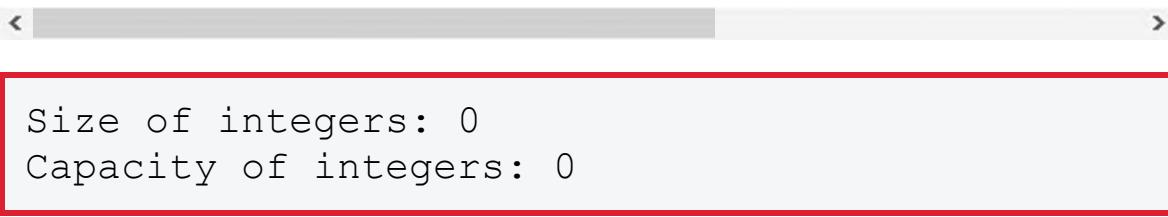
Fig. 13.2 | Standard library `vector` class template.

Creating a `vector` and Displaying Its Initial Size and Capacity

Line 15 defines the `vector<int>` object called `integers`. `vector`'s default constructor creates an empty `vector` with no elements (i.e., its size is 0) and no storage for elements (i.e., its capacity is 0, so the `vector` will have to allocate memory when elements are added to it).

[Click here to view code image](#)

```
14  int main() {  
15      std::vector<int> integers{}; // create v  
16  
17      std::cout << "Size of integers: " << integers.  
18          << "\nCapacity of integers: " << integers.  
19
```



```
Size of integers: 0  
Capacity of integers: 0
```

Lines 17–18 demonstrate the `size` and `capacity` member functions—each initially returns 0 for `integers`. Function `size`—available in every container except `forward_list`—returns the number of elements currently stored in the container. Function `capacity` (specific to `vector` and `deque`) returns `vector`'s current capacity.

`vector` Member Function `push_back`

Lines 21–24 use function `push_back` to append an element to the `vector`, then call our function `showResult` to display the item that was added to the `vector` and the new `size` and `capacity`. This function is available in sequence containers other than `array` and `forward_list`. Sequence containers other than `array` and `vector` also provide a `push_front` function. If the `vector`'s `size` equals its `capacity`, the `vector` is full so it increases its size.

[Click here to view code image](#)

```
20 // append 1-10 to integers and display update
21 for (int i : std::views::iota(1, 11)) {
22     integers.push_back(i); // push_back is implicitly
23     showResult(i, integers.size(), integers.capacity());
24 }
25
```

```
< >
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 3
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 6
appended: 6; size: 6; capacity: 6
appended: 7; size: 7; capacity: 9
appended: 8; size: 8; capacity: 9
appended: 9; size: 9; capacity: 9
appended: 10; size: 10; capacity: 13
```

Updated `size` and `capacity` After Modifying a `vector`

The manner in which a `vector` grows to accommodate more elements—a time-consuming operation—is not specified by the C++ Standard. Some implementations double the `vector`'s `capacity`. Others increase the capacity by 1.5 times, as shown in the output from **Visual C++**. This becomes apparent starting when the size and capacity are both 4:

- When we appended 5, the `vector` increased the capacity from 4 to 6, and the size became 5, leaving room for one more element.
- When we appended 6, the capacity remained at 6, and the size became 6.
- When we appended 7, the `vector` increased the capacity from 6 to 9, and the size became 7, leaving room for two more elements.
- When we appended 8, the capacity remained at 9, and the size became 8.
- When we appended 9, the capacity remained at 9, and the size became 9.

- When we appended 10, the **vector** increased the **capacity** from 9 to 13 (1.5 times 9 rounded down to the nearest integer), and the **size** became 10, leaving room for three more elements before the **vector** will need to allocate more space.

vector Growth

C++ library implementers use various schemes to minimize **vector** resizing overhead, so this program's output may vary, based on your compiler's **vector** growth implementation. **GNU g++**, for example, doubles a **vector**'s capacity when more room is needed, producing the following output:

[Click here to view code image](#)

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 4
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 8
appended: 6; size: 6; capacity: 8
appended: 7; size: 7; capacity: 8
appended: 8; size: 8; capacity: 8
appended: 9; size: 9; capacity: 16
appended: 10; size: 10; capacity: 16
```

Perf Some programmers allocate a large initial **capacity**. If a **vector** stores a small number of elements, such **capacity** may be a waste of space. However, it can greatly improve performance if a program adds many elements to a **vector** and does not have to reallocate memory to accommodate those elements. This is a classic **space-time trade-off**. Library implementors must balance the amount of memory used against the amount of time required to perform various **vector** operations.

Perf It can be wasteful to double a **vector**'s size when more space is needed. For example, in a **vector** implementation that doubles the allocated memory when space is needed, a full **vector** of 1,000,000 elements resizes

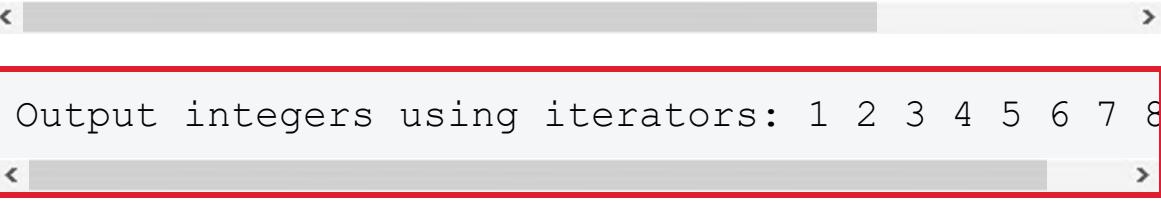
to accommodate 2,000,000 elements even when only one new element is added. This leaves 999,999 unused elements. You can use member functions **resize** and **reserve** to control space usage better.

Outputting **vector** Contents with Iterators

Lines 28–31 output the **vector**'s contents. Line 28 initializes the control variable **const-Iterator** using **vector** member function **cbegin**, which returns a **const_iterator** to the **vector**'s first element. We infer the control variable's type (**vector<int>::const_iterator**) using the **auto** keyword—this saves time and reduces errors when working with this more complex types.

[Click here to view code image](#)

```
26     std::cout << "\nOutput integers using iterators";
27
28     for (auto constIterator{integers.cbegin()});
29         constIterator != integers.cend(); ++constIterator)
30         std::cout << *constIterator << ' ';
31     }
32 }
```



```
Output integers using iterators: 1 2 3 4 5 6 7 8
```

Err  The loop continues as long as **constIterator** has not reached the end of the **vector**. This is determined by comparing **constIterator** to the result of calling the **vector**'s **cend** member function, which returns a **const_iterator** indicating the location past the last element of the **vector**. If **constIterator** is equal to this value, the end of the **vector** has been reached. **Attempting to dereference an iterator positioned outside its container is a runtime logic error—the iterator returned by **end** or **cend** should never be dereferenced or incremented.**

The loop body dereferences **constIterator** to get the current element's

value. Remember that the iterator acts like a pointer to the element and that operator `*` is overloaded to return a reference to the element. The expression `++constIterator` (line 29) positions the iterator to the `vector`'s next element. Note that you can replace this loop with the following range-based `for` statement:

[Click here to view code image](#)

```
for (auto const& item : integers) {  
    cout << item << ' ';  
}
```

The range-based `for` uses iterators “behind the scenes.”

Displaying the `vector`'s Contents in Reverse with `const_reverse_iterators`

Lines 36–39 use a `for` statement (similar to the one in `printVector`) to iterate through the `vector` in reverse. The `vector` member functions `crbegin` and `crend` each return `const_reverse_iterators` that represent the starting and ending points when iterating through a container in reverse. Most sequence containers and associative containers support this type of iterator. Class `vector` also provides member functions `rbegin` and `rend` to obtain `non-const reverse_iterators`.

[Click here to view code image](#)

```
33     std::cout << "\nOutput integers in rever  
34  
35     // display vector in reverse order using  
36     for (auto reverseIterator{integers.crbegin()  
37         ; reverseIterator != integers.crend());  
38         std::cout << *reverseIterator << ' ';  
39     }  
40  
41     std::cout << "\n";  
42 }
```



```
Output integers in reverse using iterators: 9 8
```

11 C++11: `shrink_to_fit`

As of C++11, you can call member function `shrink_to_fit` to request that a `vector` or `deque` return unneeded memory to the system, reducing its capacity to the container's current number of elements. According to the C++ standard, implementations can ignore this request so that they can perform implementation-specific optimizations.

13.6.2 `vector` Element-Manipulation Functions

Figure 13.3 illustrates functions for retrieving and manipulating `vector` elements. Line 12 initializes a `vector<int>` with the `vector` constructor that receives a braced initializer. In this case, we declared the `vector`'s type as `std::vector` and the compiler inferred the element type from the `int` values in the initializer list. Allowing the compiler to infer the element type is known as **class template argument deduction (CTAD)**. Line 13 uses a `vector` constructor that takes two iterators as arguments to initialize integers with a copy of the elements from the beginning (`values.cbegin()`) up to, but not including, the end (`values.cend()`) of the `vector` values.

[Click here to view code image](#)

```
1 // fig13_03.cpp
2 // Testing standard library vector class template
3 // element-manipulation functions.
4 #include <algorithm> // copy algorithm
5 #include <fmt/format.h> // C++20: This will
6 #include <iostream>
7 #include <ranges>
8 #include <iterator> // ostream_iterator iterator
9 #include <vector>
10
```

```
11 int main() {
12     std::vector values{1, 2, 3, 4, 5}; // c1
13     std::vector<int> integers{values.cbegin(
14         std::ostream_iterator<int> output{std::cout,
15             " "});
```



Fig. 13.3 | vector container element-manipulation functions.

ostream_iterator

Line 14 defines an **ostream_iterator** called **output** that can be used to output integers separated by single spaces via **cout**. An **ostream_iterator<int>** outputs only values of type **int** or a compatible type. The first argument to the constructor specifies the output stream, and the second argument is a string specifying the separator for the values output—in this case, the string contains a space character. We use the **ostream_iterator** (defined in header **<iterator>**) to output the contents of the **vector** in this example.

copy Algorithm

Line 17 uses standard library algorithm **copy** (from header **<algorithm>**) to output the entire contents of **integers** to the standard output. This version of the algorithm takes three arguments. The first two are iterators that specify the elements to copy from **vector** **integers**—from its beginning (**integers.cbegin()**) up to, but not including, its end (**integers.cend()**). These two arguments must satisfy **input iterator** requirements, enabling values to be read from a container, such as **const_iterators**. They must also represent elements in the same container such that applying **++** to the first iterator repeatedly eventually causes it to reach the second iterator argument. The elements are copied to the location specified by the third argument, which must be an **output iterator** through which a value can be stored or output. In this case, the output iterator is an **ostream_iterator** attached to **cout**, so the elements are copied to the standard output.

[Click here to view code image](#)

```
16     std::cout << "integers contains: ";
17     std::copy(integers.cbegin(), integers.cend(
18
```



```
integers contains: 1 2 3 4 5
```

Common Ranges

Prior to C++20, a “range” of container elements to process was described by iterators specifying the starting position and the one-past-the-end position. Such a range often is specified with calls to a container’s begin and end (or similar) member functions, as in line 17. As of C++20, the C++ standard refers to this as a **common range**, so that it is not confused by the new C++20 ranges capabilities. From this point forward, we’ll use this the term “common range” when discussing ranges determined by two iterators, and we’ll use the term “range” when discussing C++20 ranges.

vector Member Functions **front** and **back**

Err Lines 19–20 get the **vector**’s first and last elements via member functions **front** and **back**, which are available in most **sequence containers**. Notice the difference between functions **front** and **begin**. Function **front** returns a reference to the **vector**’s first element, while function **begin** returns a **random-access iterator** pointing to the **vector**’s first element. Similarly, notice the difference between **back** and **end**. Function **back** returns a reference to the **vector**’s last element, whereas **end** returns a **random-access iterator** pointing to the location after the last element. The results of **front** and **back** are undefined when called on an empty **vector**.

[Click here to view code image](#)

```
19     std::cout << fmt::format("\nfront: {}\nback
20                     integers.front(), integers.
21
```

```
front: 1  
back: 5
```

Accessing `vector` Elements

Lines 22–23 illustrate two ways to access `vector` elements. These can also be used with `deque` containers. Line 22 uses the subscript operator that's overloaded to return either a reference to the value at the specified location or a reference to that `const` value, depending on whether the container is `const`. Function `at` (line 23) performs the same operation, but with **bounds checking**. The `at` member function first checks its argument and determines whether it's in the `vector`'s bounds. If not, function `at` throws an `out_of_range` exception (which we demonstrated in Section 6.17).

[Click here to view code image](#)

```
22     integers[0] = 7; // set first element to 7  
23     integers.at(2) = 10; // set element at posi  
24
```

`vector` Member Function `insert`

Line 26 uses one of the several overloaded **insert member functions** provided by each **sequence container** (except `array`, which has a fixed size, and `forward_list`, which has the function `insert_after` instead). Line 26 inserts the value 22 before the element at the location specified by the iterator in the first argument. Here, the iterator points to the `vector`'s second element, so 22 is inserted as the second element and the original second element becomes the third element. Other versions of `insert` allow

- inserting multiple copies of the same value starting at a particular position, or
- inserting a range of values from another container, starting at a particular

position.

The version of member function **insert** in line 26 returns an iterator pointing to the item that was inserted.

[Click here to view code image](#)

```
25 // insert 22 as 2nd element
26 integers.insert(integers.cbegin() + 1, 22);
27
28 std::cout << "Contents of vector integers a
29 std::ranges::copy(integers, output);
30
```



```
< >
Contents of vector integers after changes: 7 22
< >
```

20 C++20 Ranges Algorithm **copy**

Line 29 uses the C++20 **copy algorithm** from the **std::ranges namespace** to copy the elements of integers to the standard output. This version of C++20 ranges version of **copy** receives only the range to copy and the output iterator representing where to copy the range's elements. The first argument is an **object that represents a range of elements** and has **input iterators** representing its beginning and end—in this case, a **vector**. Most of the pre-C++20 algorithms in the **<algorithm>** header now have versions in the **std** namespace and in the **std::ranges** namespace. The **std::ranges** algorithms typically are overloaded with a version that takes a **range object** and a version that takes an iterator and a sentinel. In the context of C++20 ranges, a **sentinel** is an object that represents when the end of the container has been reached. Though many of the standard library algorithms now have C++20 ranges versions, **some common-ranges algorithms were not overloaded with C++20 ranges versions for C++20**. We'll demonstrate many C++20 ranges algorithms in [Chapter 14](#).

vector Member Function **erase**

Lines 31 and 36 use the two **`erase`** member functions that are available in most **sequence containers** and **associative containers**—except **`array`**, which has a fixed size, and **`forward_list`**, which has the function **`erase_after`** instead. Line 31 erases the element at the location specified by its iterator argument—in this case, the first element. Line 36 specifies that all elements in the range specified by the two iterator arguments should be erased—in this case, all the elements are erased. Line 38 uses member function **`empty`** (available for all containers and adaptors) to confirm that the **`vector`** is empty.

[Click here to view code image](#)

```
31     integers.erase(integers.cbegin()); // erase
32     std::cout << "\n\nintegers after erasing fi
33     std::ranges::copy(integers, output);
34
35     // erase remaining elements
36     integers.erase(integers.cbegin(), integers.cend());
37     std::cout << fmt::format("\nErased all elem
38                         integers.empty() ? "is" : "
39
```

```
<                                >
integers after erasing first element: 22 2 10 4
Erased all elements: integers is empty
<                                >
```

Err  Normally `erase` destroys the objects that are erased from a container. However, **erasing an element that is a pointer to a dynamically allocated object does not delete the object, potentially causing a memory leak**. If the element is a **`unique_ptr`** (Section 11.5), the **`unique_ptr`** would be destroyed and the dynamically allocated memory would be deleted. If the element is a **`shared_ptr`** (Chapter 19), the reference count to the dynamically allocated object would be decremented and the memory would be deleted only if the reference count reached 0.

vector Member Function `insert` with Three Arguments (Range `insert`)

Line 41 demonstrates the version of function `insert` that uses the second and third arguments to specify the starting location and ending location in a sequence of values (in this case, from the `vector` `values`) that should be inserted into the `vector`. Remember that the ending location specifies the position in the sequence **after** the last element to be inserted; copying occurs up to, but not including, this location. This version of member function `insert` returns an iterator pointing to the first item that was inserted—if nothing was inserted, the function returns its first argument.

[Click here to view code image](#)

```
40 // insert elements from the vector values
41 integers.insert(integers.cbegin(), values.cbegin(),
42 std::cout << "\nContents of vector integers"
43 std::ranges::copy(integers, output);
44
```

Contents of vector integers before clear: 1 2 3

vector Member Function `clear`

Finally, line 46 uses member function `clear` to empty the `vector`. All sequence containers and associative containers except `array`, which is fixed in size, provide member function `clear`. **This does not reduce the vector's capacity.**

[Click here to view code image](#)

```
45 // empty integers; clear calls erase to empty
46 integers.clear();
47 std::cout << fmt::format("\nAfter clear, integers contains {} elements",
```

```
48         integers.empty() ? "is" : "  
49     }
```

After clear, integers is empty

13.7 `list` Sequence Container

The **`list` sequence container** (from header `<list>`) allows insertion and deletion operations at any location in the container. If most of the insertions and deletions occur at the ends of the container, the **`deque`** data structure (Section 13.8) provides a more efficient implementation of inserting at the front, which is **$O(1)$** for `deque` vs. **$O(n)$** for `vector`. The **`list`** container is implemented as a **doubly linked list**¹⁰—every node in the `list` contains a pointer to the previous node in the `list` and to the next node in the `list`. This enables `lists` to support **bidirectional iterators** that allow the container to be traversed both forward and backward. Any algorithm that requires **input**, **output**, **forward** or **bidirectional iterators** can operate on a `list`. Many `list` member functions manipulate the elements of the container as an ordered set of elements.

¹⁰. “`std::list`.” Accessed April 11, 2021.
<https://en.cppreference.com/w/cpp/container/list>.

`forward_list` Container

The **`forward_list` sequence container** (header `<forward_list>`; added in C++11) is implemented as a **singly linked list**—every node contains a pointer to the next node in the `forward_list`. This enables a `forward_list` to support **forward iterators** that allow the container to be traversed in the forward direction. Any algorithm that requires **input**, **output** or **forward iterators** can operate on a `forward_list`.

`list` Member Functions

In addition to the member functions in Fig. 11.2 and the common member functions of all **sequence containers** discussed in Section 13.5, the `list` container provides other member functions, including **`splice`**,

push_front, **pop_front**, **remove**, **remove_if**, **unique**, **merge**, **reverse** and **sort**. Several of these member functions are **list**-optimized implementations of the standard library algorithms presented in Chapter 14. Both **push_front** and **pop_front** are also supported by **forward_list** and **deque**. Figure 13.4 demonstrates several features of class **list**. Remember that many of the functions presented in Figs. 13.2–13.3 can be used with the **list** container, so we focus on the new features in this example’s discussion.

[Click here to view code image](#)

```
1 // fig13_04.cpp
2 // Standard library list class template.
3 #include <algorithm> // copy algorithm
4 #include <iostream>
5 #include <iterator> // ostream_iterator
6 #include <list> // list class-template definition
7 #include <vector>
8
9 // printList function template definition;
10 // ostream_iterator and copy algorithm to output
11 template <typename T>
12 void printList(const std::list<T>& items) {
13     if (items.empty()) { // list is empty
14         std::cout << "List is empty";
15     }
16     else {
17         std::ostream_iterator<T> output{std::cout};
18         std::ranges::copy(items, output);
19     }
20 }
21
```

Fig. 13.4 | Standard library **list** class template.

Function template `printList` (lines 11–20) checks whether its `list` argument is empty (line 13) and, if so, displays an appropriate message. Otherwise, `printList` uses an `ostream_iterator` and the `std::ranges::copy` algorithm to copy the `list`'s elements to the standard output, as shown in Fig. 13.3.

Creating a `list` Object

Line 23 creates a `list` object capable of storing `ints`. Lines 26–27 use `list` member function `push_front` to insert integers at the beginning of values. This member function is specific to classes `forward_list`, `list` and `deque`. Lines 28–29 use `push_back` to append integers to values. **Function `push_back` is common to all sequence containers, except `array` and `forward_list`.** Lines 31–32 show the current contents of values.

[Click here to view code image](#)

```
22 int main() {  
23     std::list<int> values{}; // create list  
24  
25     // insert items in values  
26     values.push_front(1);  
27     values.push_front(2);  
28     values.push_back(4);  
29     values.push_back(3);  
30  
31     std::cout << "values contains: ";  
32     printList(values);  
33
```



`list` Member Function `sort`

Line 34 uses `list` member function `sort` to arrange the elements in the

list in ascending order. A second version of function **sort** allows you to supply a **binary predicate function** that takes two arguments (values in the **list**), performs a comparison and returns a **bool** value indicating whether the first argument should come before the second in the sorted contents. This function determines the order in which the elements of the **list** are sorted. This version could be particularly useful for a **list** that stores pointers rather than values.

[Click here to view code image](#)

```
34     values.sort(); // sort values
35     std::cout << "\nvalues after sorting contains: ";
36     printList(values);
37
38
```

```
values after sorting contains: 1 2 3 4
```

list Member Function **splice**

Line 37 uses **list** member function **splice** to remove the elements in **otherValues** and insert them into **values** before the iterator position specified as the first argument. Function **splice** with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument. Function **splice** with four arguments uses the last two arguments to specify a range of locations that should be removed from the container in the second argument and placed at the location specified in the first argument. A **forward_list** provides a similar member function named **splice_after**.

[Click here to view code image](#)

```
39 // insert elements of ints into otherValues
```

```
40     std::vector ints{2, 6, 4, 8};
41     std::list<int> otherValues{}; // create list
42     otherValues.insert(otherValues.cbegin(), in
43     std::cout << "\nAfter insert, otherValues contains: ";
44     printList(otherValues);
45
46     // remove otherValues elements and insert a
47     values.splice(values.cend(), otherValues);
48     std::cout << "\nAfter splice, values contains: ";
49     printList(values);
```



```
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
```

list Member Function `merge`

After inserting more elements in `otherValues` and sorting both `values` and `otherValues`, line 61 uses `list` member function `merge` to remove all elements of `otherValues` and insert them in sorted order into `values`. Both `lists` must be sorted in the same order before this operation is performed. A second version of `merge` enables you to supply a **binary predicate function** that takes two arguments (values in the `list`) and returns a `bool` value. The predicate function specifies the sorting order used by `merge`.

[Click here to view code image](#)

```
50     values.sort(); // sort values
51     std::cout << "\nAfter sort, values contains: ";
52     printList(values);
53
54     // insert elements of ints into otherValues
55     otherValues.insert(otherValues.cbegin(), in
56     otherValues.sort(); // sort the list
57     std::cout << "\nAfter insert and sort, otherValues contains: "
```

```
58     printList(otherValues);
59
60     // remove otherValues elements and insert in values
61     values.merge(otherValues);
62     std::cout << "\nAfter merge:\n values contains: ";
63     printList(values);
64     std::cout << "\n otherValues contains: ";
65     printList(otherValues);
66
```

```
<          >
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4
    values contains: 1 2 2 2 3 4 4 4 6 6 8 8
    otherValues contains: List is empty
<          >
```

[list Member Function `pop_front` and `pop_back`](#)

Line 67 uses `list` member function `pop_front` to remove the first element in the `list`. Line 68 uses function `pop_back` to remove the last element in the `list`. This function is available for **sequence containers** other than `array` and `forward_list`.

[Click here to view code image](#)

```
67     values.pop_front(); // remove element from front
68     values.pop_back(); // remove element from back
69     std::cout << "\nAfter pop_front and pop_back:\n";
70     printList(values);
71
```

```
<          >
After pop_front and pop_back:
values contains: 2 2 2 3 4 4 4 6 6 8
```

list Member Function **unique**

Line 72 uses **list** function **unique** to **remove duplicate elements** in the **list**. The **list** should be in sorted order (so that all duplicates are side by side) before this operation is performed, to **guarantee that all duplicates are eliminated**. A second version of **unique** enables you to supply a **predicate function** that takes two arguments (values in the **list**) and returns a **bool** value specifying whether two elements are equal.

[Click here to view code image](#)

```
72     values.unique(); // remove duplicate elements
73     std::cout << "\nAfter unique, values contains: ";
74     printList(values);
75
```

```
<          >
After unique, values contains: 2 3 4 6 8
```

list Member Function **swap**

Line 76 uses **list** member function **swap** to exchange the contents of **values** with the contents of **otherValues**. This function is available to **all sequence containers and associative containers**.

[Click here to view code image](#)

```
76     values.swap(otherValues); // swap elements
77     std::cout << "\nAfter swap:\n values contains: ";
78     printList(values);
79     std::cout << "\n otherValues contains: ";
80     printList(otherValues);
81
```

```
<          >
After swap:
```

```
values contains: List is empty
otherValues contains: 2 3 4 6 8
```

list Member Functions `assign` and `remove`

Line 83 uses `list` member function `assign` (available to all **sequence containers**) to replace the contents of `values` with the contents of `otherValues` in the range specified by the two iterator arguments. A second version of `assign` replaces the original contents with copies of the value specified in the second argument. The first argument of the function specifies the number of copies. Line 92 uses `list` member function `remove` to delete all copies of the value 4 from the `list`.

[Click here to view code image](#)

```
82      // replace contents of values with elements of otherValues
83      values.assign(otherValues.cbegin(), otherValues.cend());
84      std::cout << "\nAfter assign, values contains: ";
85      printList(values);
86
87      // remove otherValues elements and insert them into values
88      values.merge(otherValues);
89      std::cout << "\nAfter merge, values contains: ";
90      printList(values);
91
92      values.remove(4); // remove all 4s
93      std::cout << "\nAfter remove(4), values contains: ";
94      printList(values);
95      std::cout << "\n";
96 }
```

```
< >
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8
After remove(4), values contains: 2 2 3 3 6 6 8
```

13.8 **deque** Sequence Container

Perf  Class **deque** provides many of the benefits of a **vector** and a **list** in one container. The term **deque** is short for “double-ended queue.” Class **deque** is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a **vector**. Class **deque** is also implemented for **efficient insertion and deletion operations at its front and back**, much like a **list** (although a **list** is also capable of efficient insertions and deletions in the middle). Class **deque** provides support for **random-access iterators**, so **deques** can be used with all standard library algorithms. One of the most common uses of a **deque** is to maintain a first-in, first-out queue of elements. In fact, a **deque** is the default underlying implementation for the **queue** adaptor¹¹ (Section 13.10.2).

11. “std::queue.” Accessed April 11, 2021.
<https://en.cppreference.com/w/cpp/container/queue>.

Additional storage for a **deque** can be allocated at either end in blocks of memory that are typically maintained as a **built-in array of pointers to those blocks**.¹² Due to a **deque’s noncontiguous memory layout**, its iterators must be more “intelligent” than the pointers that are used to iterate through **vectors**, **arrays** or built-in arrays. A **deque** provides the same basic operations as **vector**, but like **list** adds member functions **push_front** and **pop_front** for efficient insertion and deletion at the beginning of the **deque**.

12. This is an implementation-specific detail, not a requirement of the C++ standard.

Figure 13.5 demonstrates features of class **deque**. Remember that many of the functions presented in Figs. 13.2–Fig. 13.4 also can be used with class **deque**. Header **<deque>** must be included to use class **deque**.

[Click here to view code image](#)

```
1 // fig13_05.cpp
2 // Standard library deque class template.
3 #include <algorithm> // copy algorithm
```

```
4 #include <deque> // deque class-template de
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7
8 int main() {
9     std::deque<double> values; // create deque
10    std::ostream_iterator<double> output{std::cout};
11
12    // insert elements in values
13    values.push_front(2.2);
14    values.push_front(3.5);
15    values.push_back(1.1);
16
17    std::cout << "values contains: ";
18
19    // use subscript operator to obtain element
20    for (size_t i{0}; i < values.size(); ++i)
21        std::cout << values[i] << ' ';
22    }
23
24    values.pop_front(); // remove first element
25    std::cout << "\nAfter pop_front, values contains: ";
26    std::ranges::copy(values, output);
27
28    // use subscript operator to modify element
29    values[1] = 5.4;
30    std::cout << "\nAfter values[1] = 5.4, values contains: ";
31    std::ranges::copy(values, output);
32    std::cout << "\n";
33 }
```



```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

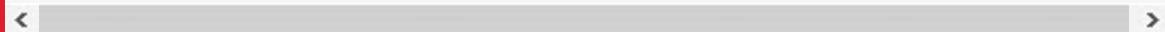


Fig. 13.5 | Standard library deque class template.

Line 9 instantiates a **deque** that can store `double` values. Lines 13–15 use member functions **push_front** and **push_back** to insert elements at the beginning and end of the **deque**.

Lines 20–22 use the subscript operator to retrieve the value in each element of the **deque** for output. The condition uses member function **size** to ensure that we do not attempt to access an element outside the bounds of the **deque**. We use a counter-controlled `for` loop here only to demonstrate the `[]` operator—generally, you should use the range-based `for` to process all the elements of a container.

Line 24 uses member function **pop_front** to demonstrate removing the first element of the **deque**. Line 29 uses the subscript operator to obtain an *lvalue*, which we use to assign a new value to element 1 of the **deque**.

13.9 Associative Containers

The **associative containers** provide **direct access** to store and retrieve elements via **keys** (also called **search keys**). The four **ordered associative containers** are **multiset**, **set**, **multimap** and **map**. Each of these maintains its keys in sorted order. There are also four corresponding **unordered associative containers**—**unordered_multiset**, **unordered_set**, **unordered_multimap** and **unordered_map**—that offer the most of the same capabilities as their ordered counterparts. The primary difference between the ordered and unordered associative containers is that the unordered ones do not maintain their keys in sorted order. In this section, we focus on the **ordered associative containers**.

Perf  For cases in which it's not necessary to maintain keys in sorted order, the **unordered associative containers** offer better search performance via **hashing**— $O(1)$ with a worst case of $O(n)$ vs. $O(\log n)$ for the ordered associative containers.¹³ For an introduction to hashing, see [Section 13.13](#).

13. “Containers library.” Accessed April 16, 2021. <https://en.cppreference.com/w/cpp/container>.

17 Iterating through an **ordered associative container** traverses it in the sort

order for that container. Classes `multiset` and `set` provide operations for manipulating sets of values where the values themselves are the keys. **The primary difference between a `multiset` and a `set` is that a `multiset` allows duplicate keys and a `set` does not.** Classes `multimap` and `map` provide operations for manipulating values associated with keys (these values are sometimes referred to as **mapped values**). **The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.** In addition to the common container member functions, **ordered associative containers** also support several member functions that are specific to associative containers. Like `list` and `forward_list`, you can combine the contents of associative containers of the same type using the `merge` function (added in C++17). Examples of each of the **ordered associative containers** and their common member functions are presented in the next several subsections.

13.9.1 `multiset` Associative Container

The **`multiset` ordered associative container** (from header `<set>`) provides fast storage and retrieval of keys and allows duplicate keys. The elements' ordering is determined by a **comparator function object**. A **function object** is an instance of a class that has an overloaded parentheses operator, allowing the object to be “called” like a function. For example, in an integer `multiset`, elements can be sorted in ascending order by ordering the keys with **comparator function object `less<int>`**, which knows how to compare two `int` values to determine whether the first is less than the second. This enables an integer `multiset` to order its elements in ascending order. We discuss function objects in detail in [Section 14.5](#). For this chapter, we'll simply show how to use `less<int>` when declaring ordered associative containers.

The data type of the keys in all **ordered associative containers** must support comparison based on the **comparator function object**—keys sorted with `less<T>` must support comparison with `operator<`. If the keys used in the **ordered associative containers** are of user-defined data types, those types must supply the appropriate comparison operators. A `multiset` supports **bidirectional iterators** (but not **random-access iterators**). If the order of the keys is not important, use `unordered_multiset` (header

`<unordered_set>.`

Creating a `multiset`

Figure 13.6 demonstrates the `multiset` ordered associative container with `int` keys that are sorted in ascending order. Containers `multiset` and `set` (Section 13.9.2) provide the same basic functionality. Line 12 creates the `multiset`, using the function object `less<int>` to specify the keys' sort order. Ascending order is the default for a `multiset`, so line 12 can be written as

[Click here to view code image](#)

```
std::multiset<int> ints{}; // multiset of int values
```



[Click here to view code image](#)

```
1 // fig13_06.cpp
2 // Standard library multiset class template
3 #include <algorithm> // copy algorithm
4 #include <fmt/format.h> // C++20: This will
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <ranges>
8 #include <set> // multiset class-template defn
9 #include <vector>
10
11 int main() {
12     std::multiset<int, std::less<int>> ints{
```



Fig. 13.6 | Standard library `multiset` class template.

`multiset` Member Function `count`

Line 13 uses function `count` (available to all **associative containers**) to count the number of occurrences of the value 15 currently in the `multiset`.

[Click here to view code image](#)

```
13     std::cout << fmt::format("15s in ints: {}\\n"
14
```

< >

```
15s in ints: 0
```

multiset Member Function **insert**

Lines 16–17 use one of the several overloaded versions of member function **insert** to add the value 15 to the **multiset** twice. A second version of **insert** takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified. A third version of **insert** takes two iterators as arguments that specify a range of values to add to the **multiset** from another container. For several other overloads, see

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/container/multiset/insert>

< >

[Click here to view code image](#)

```
15     std::cout << "\\nInserting two 15s into ints
16     ints.insert(15); // insert 15 in ints
17     ints.insert(15); // insert 15 in ints
18     std::cout << fmt::format("15s in ints: {}\\n"
19
```

< >

```
Inserting two 15s into ints
15s in ints: 2
```

multiset Member Function **find**

Lines 21–28 use member function **find** (line 22), which is available to all **associative containers**, to search for the values 15 and 20 in the **multiset**. The range-based **for** loop iterates through each item in {15, 20}. Function **find** returns either an **iterator** or a **const_iterator**, depending on whether the **multiset** is const. The iterator points to the location at which the value is found. If the value is not found, **find** returns an **iterator** or a **const_iterator** equal to the value returned by the container's **end** member function.

[Click here to view code image](#)

```
20 // search for 15 and 20 in ints; find returns an iterator or const_iterator
21 for (int i : {15, 20}) {
22     if (auto result{ints.find(i)}; result != ints.end())
23         std::cout << fmt::format("Found {} in ints\n", *result);
24     else
25         std::cout << fmt::format("Did not find {}\n", i);
26 }
27 }
28 }
29 }
```



```
Found 15 in ints
Did not find 20 in ints
```

20 **multiset** Member Function **contains** (C++20)

Lines 31–39 use the new C++20 member function **contains** (line 22) to determine whether the values 15 and 20 are in the **multiset**. This function, which is available to all **associative containers**, returns a **bool** indicating whether the value is present in the container. The range-based **for** loop iterates through each item in {15, 20}. Function **find** returns either an **iterator** or a **const_iterator**, depending on whether the **multiset** is const. The iterator points to the location at which the value

is found. If the value is not found, **find** returns an **iterator** or a **const_iterator** equal to the value returned by the container's **end** member function.

[Click here to view code image](#)

```
30 // search for 15 and 20 in ints; contains r
31 for (int i : {15, 20}) {
32     if (ints.contains(i)) {
33         std::cout << fmt::format("Found {} in
34     }
35     else {
36         std::cout << fmt::format("Did not fin
37     }
38 }
39
```

< >

```
Found 15 in ints
Did not find 20 in ints
```

[Inserting Elements of Another Container into a multiset](#)

Line 42 uses member function **insert** to insert a **vector**'s elements into the **multiset**, then line 44 copies the **multiset**'s elements to the standard output. The values display in ascending order because the **multiset** is an ordered container that maintains its elements in ascending order by default. In line 44, note the expression:

[Click here to view code image](#)

```
std::ostream_iterator<int>{std::cout, " "}
```

This creates an **ostream_iterator** and immediately passes it to the **copy algorithm**. We chose this approach because the **ostream_iterator** is used only once in this example.

[Click here to view code image](#)

```
40 // insert elements of vector values into int
41 const std::vector values{7, 22, 9, 1, 18, 3
42 ints.insert(values.cbegin(), values.cend())
43 std::cout << "\nAfter insert, ints contains"
44 std::ranges::copy(ints, std::ostream_iterator<
45
```



After insert, ints contains:

```
1 7 9 13 15 15 18 22 22 30 85 100
```

multiset Member Functions `lower_bound` and `upper_bound`

Line 49 uses functions **`lower_bound`** and **`upper_bound`**, available in all **associative containers**, to locate the earliest occurrence of the value 22 in the **`multiset`** and the element after the last occurrence of the value 22 in the **`multiset`**. Both functions return **iterators** or **const_iterators** pointing to the appropriate location or the iterator returned by `end` if the value is not in the **`multiset`**. Together, the lower bound and upper bound represent the **common range** of elements containing the value 22.

[Click here to view code image](#)

```
46 // determine lower and upper bound of 22 in
47 std::cout << fmt::format(
48             "\n\nlower_bound(22): {}\\nupper_
49             * (ints.lower_bound(22)), * (
50
```



lower_bound(22): 22

upper_bound(22): 30

pair Objects and multiset Member Function equal_range

Line 52 creates and initializes a **pair** object called p. We use `auto` to infer the variable's type from its initializer—in this case, the return value of **multiset** member function **equal_range**, which is a **pair** object. Such objects associate pairs of values. In this case, the contents of p will be two **iterators** or **const_iterators**, depending on whether the **multiset** is `const`. The **multiset** function **equal_range** returns a pair containing the results of calling both **lower_bound** and **upper_bound**. Class template **pair** contains two public data members called **first** and **second**—their types depend on the **pair**'s initializers.

[Click here to view code image](#)

```
51 // use equal_range to determine lower and
52 auto p{ints.equal_range(22)};
53 std::cout << fmt::format(
54     "lower_bound(22): {}\\nupper_
55             * (p.first), * (p.second) );
56 }
```

< >

```
lower_bound(22): 22
upper_bound(22): 30
```

Line 52 uses function **equal_range** to determine the **lower_bound** and **upper_bound** of 22 in the **multiset**. Line 55 uses `p.first` and `p.second` to access the **lower_bound** and **upper_bound**. We dereferenced the iterators to output the values at the locations returned from **equal_range**. Though we did not do so here, you should always ensure that the iterators returned by **lower_bound**, **upper_bound** and **equal_range** are not equal to the container's end iterator before dereferencing the iterators.

Variadic Class Template tuple

C++ also includes class template **tuple** (added in C++11), which is similar

to **pair**, but can hold any number of items of various types. Class template **tuple** is implemented using **variadic templates**—templates that can receive a variable number of arguments. We discuss **tuple** and variadic templates in [Chapter 15, Templates, C++20 Concepts and Metaprogramming](#).

14 C++14: Heterogeneous Lookup

Prior to C++14, when searching for a key in an associative container, the argument provided to a search function like **find** was required to have the container’s key type. For example, if the key type were **string**, you could pass **find** a pointer-based string to locate in the container. In this case, the argument would be converted into a temporary object of the key type (**string**), then passed to **find**. In C++14 and higher, the argument to **find** (and other similar functions) can be of any type, provided that there are overloaded comparison operators that can compare values of the argument’s type to values of the container’s key type. If there are, no temporary objects will be created. This is known as **heterogeneous lookup**.

13.9.2 **set** Associative Container

The **set** associative container (from header `<set>`) is used for fast storage and retrieval of unique keys. The implementation of a **set** is identical to that of a **multiset**, except that a **set** must have unique keys. Therefore, if an attempt is made to insert a duplicate key into a **set**, the duplicate is ignored—this is the intended mathematical behavior of a set, so it’s not considered an error. A **set** supports **bidirectional iterators** (but not **random-access iterators**). If the order of the keys is not important, you can use **unordered_set** (header `<unordered_set>`) instead.

Creating a **set**

[Figure 13.7](#) demonstrates a **set** of doubles. Line 10 creates the **set**, using class template argument deduction (CTAD) to infer the **set**’s type. Line 10 is equivalent to:

[Click here to view code image](#)

```
std::set<double, std::less<double>> doubles{2.1, .
```



20 The **set**'s **initializer_list** constructor inserts all the elements into the **set**. Line 14 uses the **std::ranges algorithm copy** to output the **set**'s contents. Notice that the value **2.1**, which appeared twice in the initializer list, appears only once in **doubles**, because **a set does not allow duplicates**.

[Click here to view code image](#)

```
1 // fig13_07.cpp
2 // Standard library set class template.
3 #include <algorithm>
4 #include <fmt/format.h> // C++20: This will
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <set>
8
9 int main() {
10     std::set doubles{2.1, 4.2, 9.5, 2.1, 3.7
11
12     std::ostream_iterator<double> output{std
13     std::cout << "doubles contains: ";
14     std::ranges::copy(doubles, output);
15 }
```

< >

```
doubles contains: 2.1 3.7 4.2 9.5
```

Fig. 13.7 | Standard library **set** class template.

Inserting a New Value into a **set**

Line 19 defines and initializes a **pair** to store the result of calling **set** member function **insert**. The **pair** contains an **iterator** pointing to the item in the **set** and a **bool** indicating whether the item was inserted —true if the item was not previously in the **set** and **false** otherwise. In this case, line 19 uses function **insert** to place the value **13.8** in the **set**.

and returns a **pair** in which p.first points to the value 13.8 in the **set** and p.second is true because the value was inserted.

[Click here to view code image](#)

```
16 // insert 13.8 in doubles; insert returns p
17 // p.first represents location of 13.8 in d
18 // p.second represents whether 13.8 was ins
19 auto p{doubles.insert(13.8)}; // value not
20 std::cout << fmt::format("\n{} {} inserted\n",
21                         (p.second ? "was" : "was no
22 std::cout << "doubles contains: ";
23 std::ranges::copy(doubles, output);
24
```

< >

```
13.8 was inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

Inserting an Existing Value into a **set**

Line 26 attempts to insert 9.5, which is already in the **set**. The output shows that 9.5 was not inserted because **sets don't allow duplicate keys**. In this case, p.first in the returned **pair** points to the existing 9.5 in the **set** and p.second is false.

[Click here to view code image](#)

```
25 // insert 9.5 in doubles
26 p = doubles.insert(9.5); // value already
27 std::cout << fmt::format("\n{} {} inserted\n",
28                         (p.second ? "was" : "was not
29 std::cout << "doubles contains: ";
30 std::ranges::copy(doubles, output);
31 std::cout << "\n";
```

```
32 }
```

```
< >
9.5 was not inserted doubles contains:
2.1 3.7 4.2 9.5 13.8
```

13.9.3 **multimap** Associative Container

The **multimap associative container** is used for fast storage and retrieval of keys and associated values (often called **key-value pairs**). Many of the functions used with **multisets** and **sets** are also used with **multimaps** and **maps**. The elements of **multimaps** and **maps** are **pairs** of keys and values instead of individual values. When inserting into a **multimap** or **map**, you use a **pair** object containing the key and the value. The ordering of the keys is determined by a **comparator function object**. For example, in a **multimap** that uses integers as the key type, keys can be sorted in **ascending order** by ordering them with **comparator function object less<int>**.

Duplicate keys are allowed in a **multimap**, so multiple values can be associated with a single key. This is called a **one-to-many relationship**. For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many students; in the military, one rank (like “private”) has many people. A **multimap** supports **bidirectional iterators**, but not **random-access iterators**.

Creating a **multimap** Containing Key–Value Pairs

Perf  Figure 13.8 demonstrates the **multimap associative container**. Header `<map>` must be included to use class **multimap**. If the order of the keys is not important, you can use **unordered_multimap** (header `<unordered_map>`) instead. A **multimap** is implemented to efficiently locate all values paired with a given key. Line 8 creates a **multimap** in which the key type is `int`, the type of a key’s associated value is `double` and the elements are ordered in ascending order by default. This is equivalent to:

[Click here to view code image](#)

```
std::multimap<int, double, std::less<int>> pairs{
```



[Click here to view code image](#)

```
1 // fig13_08.cpp
2 // Standard library multimap class template
3 #include <fmt/format.h> // C++20: This will
4 #include <iostream>
5 #include <map> // multimap class-template de-
6
7 int main() {
8     std::multimap<int, double, std::less<int>>

```



Fig. 13.8 | Standard library multimap class template.

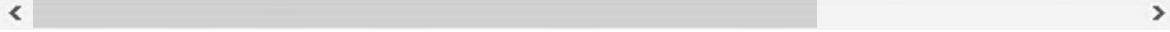
Counting the Number of Key–Value Pairs for a Specific Key

Line 10 uses member function **count** to determine the number of key–value pairs with a key of 15—in this case, 0 since the container is currently empty.

[Click here to view code image](#)

```
9     std::cout << fmt::format("Number of 15 keys
10                     pairs.count(15));
11

```



```
There are currently 0 pairs with key 15 in the m
```



Inserting Key–Value Pairs

Line 13 uses member function **insert** to add a new key–value pair to the

multimap. Standard library function **make_pair** creates a **pair** object—in this case first represents a key (15) of type `int` and second represents a value (99.3) of type `double`. Function **make_pair** automatically uses the types that you specified for the keys and values in the **multimap**'s declaration (line 8). Line 14 inserts another **pair** object with the key 15 and the value 2.7. Then lines 15–16 output the number of **pairs** with key 15.

[Click here to view code image](#)

```
12 // insert two pairs
13 pairs.insert(std::make_pair(15, 99.3));
14 pairs.insert(std::make_pair(15, 2.7));
15 std::cout << fmt::format("Number of 15 keys
16                 pairs.count(15));
17
```

< >

After inserts, there are 2 pairs with key 15

Inserting Key–Value Pairs with Braced Initializers Rather Than **make_pair**

You can use braced initialization for **pair** objects, so lines 13–14 can be simplified as

```
pairs.insert({15, 99.3});
pairs.insert({15, 2.7});
```

Similarly, you can use braced initialization to initialize an object being returned from a function. For example, the following returns a **pair** containing an `int` and a `double`

```
return {15, 2.7};
```

Lines 19–23 insert five additional **pairs** into the **multimap**. The range-based **for** statement in lines 28–30 outputs the **multimap**'s keys and values. We infer the type of the loop's control variable—in this case, a **pair**

containing an `int` key and a `double` value. Line 29 accesses each `pair`'s members. Notice that the keys appear in ascending order because the `multimap` maintains the keys in ascending order.

[Click here to view code image](#)

```
18     // insert five pairs
19     pairs.insert({30, 111.11});
20     pairs.insert({10, 22.22});
21     pairs.insert({25, 33.333});
22     pairs.insert({20, 9.345});
23     pairs.insert({5, 77.54});
24
25     std::cout << "Multimap pairs contains:\n";
26
27     // walk through elements of pairs
28     for (const auto& mapItem : pairs) {
29         std::cout << fmt::format("{}\t{}\n", 1
30     }
31 }
```



Multimap pairs contains:

Key	Value
5	77.54
10	22.22
15	99.3
15	2.7
20	9.345
25	33.333
30	111.11

11 C++11: Brace Initializing a Key–Value Pair Container

This example used separate calls to member function `insert` to place key–

value pairs in a **multimap**. If you know the key–value pairs in advance, you can use braced initialization when you create the **multimap**. For example, the following statement initializes a **multimap** with three key–value **pairs** that are represented by the sublists in the main initializer list:

[Click here to view code image](#)

```
std::multimap<int, double> pairs{  
    {10, 22.22}, {20, 9.345}, {5, 77.54}};
```

13.9.4 map Associative Container

The **map associative container** (from header `<map>`) performs fast storage and retrieval of unique keys and associated values. Duplicate keys are not allowed—a single value can be associated with each key. This is called a **one-to-one mapping**. For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a **map** that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively. Providing the key in a **map**’s subscript operator `[]` locates the value associated with that key in the **map**. Insertions and deletions can be made anywhere in a **map**. If the order of the keys is not important, you can use **unordered_map** (header `<unordered_map>`) instead.

Figure 13.9 demonstrates a **map** (created at lines 8–9) and uses many of the same features as Fig. 13.8. Note that only six of the initial eight key–value pairs are inserted because two have duplicate keys. Unlike similar data structures in some programming languages, inserting two key–value pairs with the same key does not replace first key’s value with that of the second. If you insert a key–value pair for which the key is already in the **map**, the key–value pair is ignored.

[Click here to view code image](#)

```
1 // fig13_09.cpp  
2 // Standard library class map class template  
3 #include <iostream>  
4 #include <fmt/format.h> // C++20: This will  
5 #include <map> // map class-template defini
```

```
6
7 int main() {
8     // create a map; duplicate keys are igno
9     std::map<int, double> pairs{{15, 2.7}, {
10    {10, 22.22}, {25, 33.333}, {5, 77.54}
11
12    // walk through elements of pairs
13    std::cout << "pairs contains:\nKey\tValue"
14    for (const auto& pair : pairs) {
15        std::cout << fmt::format("{}\t{}\n", [
16    }
17
18    pairs[25] = 9999.99; // use subscripting
19    pairs[40] = 8765.43; // use subscripting
20
21    // use const_iterator to walk through el
22    std::cout << "\nAfter updates, pairs con
23    for (const auto& pair : pairs) {
24        std::cout << fmt::format("{}\t{}\n", [
25    }
26 }
```



```
pairs contains:
Key      Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After updates, pairs contains:
Key      Value
5       1010.1
10      22.22
15      2.7
20      9.345
```

25	9999.99
30	111.11
40	8765.43

Fig. 13.9 | Standard library map class template.

Lines 18–19 use the `map` subscript operator. When the subscript is a key that's in the `map`, the operator returns a reference to the associated value. When the subscript is a key that's not in the `map`, the subscript operator inserts a new key–value pair in the `map`, consisting of the specified key and the default value for the container's value type. Line 18 replaces the value for the key 25 (previously 33.333, as specified in line 10) with a new value, 9999.99. Line 19 inserts a new key–value **pair** in the `map`. This is known as **creating an association**.

13.10 Container Adaptors

The three **container adaptors** are `stack`, `queue` and `priority_queue`. Container adaptors do not provide a data-structure implementation in which elements can be stored and **do not support iterators**. With an adaptor class, you can choose the underlying sequence container or use the adaptor's default choice—`deque` for `stacks` and `queues`, or `vector` for `priority_queue`. The adaptor classes provide member functions `push` and `pop`:

- `push` properly inserts an element into an adaptor's underlying container.
- `pop` properly removes an element from an adaptor's underlying container.

Let's see examples of the adaptor classes.

13.10.1 `stack` Adaptor

Class `stack` (from header `<stack>`) enables insertions into and deletions from the underlying container at one end called the `top`, so a `stack` is commonly referred to as a **last-in, first-out** data structure. A `stack` can be implemented with a `vector`, `list` or `deque`. By default, a `stack` is implemented with a `deque`.¹⁴ The `stack` operations are:

14. “std::stack.” Accessed April 11, 2021.
<https://en.cppreference.com/w/cpp/container/stack>.

- **push** to insert an element at the **stack**’s **top**—implemented by calling the underlying container’s **push_back** member function,
- **pop** to remove the **stack**’s **top** element—implemented by calling the underlying container’s **pop_back** member function,
- **top** to get a reference to the **stack**’s top element—implemented by calling the underlying container’s **back** member function,
- **empty** to determine whether the **stack** is empty—implemented by calling the underlying container’s **empty** member function, and
- **size** to get the **stack**’s number of elements—implemented by calling the underlying container’s **size** member function.

Figure 13.10 demonstrates the **stack adaptor class**. To demonstrate that the **stack** adaptor works with each **sequence container**, this example creates three **stacks** of **ints**, using a **deque** (line 26), a **vector** (line 27) and a **list** (line 28) as the underlying data structure.

[Click here to view code image](#)

```
1 // fig13_10.cpp
2 // Standard library stack adaptor class.
3 #include <iostream>
4 #include <list> // list class-template defi:
5 #include <ranges>
6 #include <stack> // stack adaptor definitio:
7 #include <vector> // vector class-template
8
9 // pushElements generic lambda to push valu
10 auto pushElements = [] (auto& stack) {
11     for (auto i : std::views::iota(0, 10)) {
12         stack.push(i); // push element onto s
13         std::cout << stack.top() << ' '; // v
14     }
15 };
```

```
16
17 // popElements generic lambda to pop elements
18 auto popElements = [](auto& stack) {
19     while (!stack.empty()) {
20         std::cout << stack.top() << ' ';
21         stack.pop(); // remove top element
22     }
23 };
24
25 int main() {
26     std::stack<int> dequeStack{}; // uses a deque
27     std::stack<int, std::vector<int>> vectorStack;
28     std::stack<int, std::list<int>> listStack;
29
30     // push the values 0-9 onto each stack
31     std::cout << "Pushing onto dequeStack: ";
32     pushElements(dequeStack);
33     std::cout << "\nPopping from dequeStack: ";
34     popElements(dequeStack);
35     std::cout << "\nPopping from vectorStack: ";
36     pushElements(vectorStack);
37     std::cout << "\nPopping from vectorStack: ";
38     popElements(vectorStack);
39     std::cout << "\nPopping from listStack: ";
40     pushElements(listStack);
41     std::cout << "\nPopping from listStack: ";
42     popElements(listStack);
43     std::cout << "\nPopping from listStack: ";
44     popElements(listStack);
45     std::cout << "\n";
46 }
```



```
Pushing onto dequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto vectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto listStack: 0 1 2 3 4 5 6 7 8 9
```

```
Popping from dequeStack: 9 8 7 6 5 4 3 2 1 0  
Popping from vectorStack: 9 8 7 6 5 4 3 2 1 0  
Popping from listStack: 9 8 7 6 5 4 3 2 1 0
```

Fig. 13.10 | Standard library stack adaptor class. (Part 2 of 2.)

The generic lambda `pushElements` (lines 10–15) pushes the values 0–9 onto a `stack`. Lines 32, 34 and 36 call this lambda for each `stack`. Line 12 uses function `push` (available in each adaptor class) to place an integer on top of the `stack`. Line 13 uses `stack` function `top` to retrieve the `stack`'s `top` element for output. **Function `top` does not remove the top element.**

Function `popElements` (lines 18–23) pops the elements off each `stack`. Lines 40, 42 and 44 call this lambda for each `stack`. Line 20 uses `stack` function `top` to retrieve the `stack`'s `top` element for output. Line 21 uses function `pop`, available in each adaptor class, to remove the `stack`'s `top` element. **Function `pop` does not return a value. So, you must call `top` to obtain the top element's value before you pop that element from the stack.**

13.10.2 `queue` Adaptor

A queue is similar to a waiting line. The item in the queue the longest is the next one removed. So, a queue is referred to as a **first-in, first-out (FIFO)** data structure. Class `queue` (from header `<queue>`) enables insertions only at the `back` of the underlying data structure and deletions only from the `front`. A `queue` stores its elements in objects of the `list` or `deque` sequence containers—`deque` is the default. The common `queue` operations are:

- **push** to insert an element at the `queue`'s back—implemented by calling the underlying container's `push_back` member function,
- **pop** to remove the element at the `queue`'s front—implemented by calling the underlying container's `pop_front` member function,
- **front** to get a reference to the `queue`'s first element—implemented by calling the underlying container's `front` member function,

- **back** to get a reference to the **queue**'s last element—implemented by calling the underlying container's **back** member function,
- **empty** to determine whether the **queue** is empty—this calls the underlying container's **empty** member function, and
- **size** to get the **queue**'s number of elements—this calls the underlying container's **size** member function.

Figure 13.11 demonstrates the **queue** adaptor class. Line 7 instantiates a **queue** of doubles. Lines 10–12 use function **push** to add elements to the **queue**. The while statement in lines 17–20 uses function **empty** (available in all containers) to determine whether the **queue** is empty (line 17). While there are more elements in the **queue**, line 18 uses **queue** function **front** to read (but not remove) the **queue**'s first element for output. Line 19 removes the **queue**'s first element with function **pop**, available in all adaptor classes.

[Click here to view code image](#)

```

1 // fig13_11.cpp
2 // Standard library queue adaptor class template
3 #include <iostream>
4 #include <queue> // queue adaptor definition
5
6 int main() {
7     std::queue<double> values{}; // queue with no elements
8
9     // push elements onto queue values
10    values.push(3.2);
11    values.push(9.8);
12    values.push(5.4);
13
14    std::cout << "Popping from values: ";
15
16    // pop elements from queue
17    while (!values.empty()) {
18        std::cout << values.front() << ' ';
```

```
19         values.pop(); // remove element
20     }
21
22     std::cout << "\n";
23 }
```

```
< >
Popping from values: 3.2 9.8 5.4
```

Fig. 13.11 | Standard library queue adaptor class template. (Part 2 of 2.)

13.10.3 priority_queue Adaptor

Class **priority_queue** (from header `<queue>`) provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. By default, a **priority_queue** stores its elements in a **vector**.¹⁵ Elements added to a **priority_queue** are inserted in **priority order**, such that the highest-priority element (i.e., the largest value) will be the first element removed. This is usually accomplished by arranging the elements in a data structure called a **heap**—not to be confused with the heap for dynamically allocated memory—that always maintains its largest value (i.e., highest-priority element) at the front of the data structure. Element comparisons are performed with **comparator function object less<T>** by default.

15. “std::priority_queue.” Access April 15, 2021.
https://en.cppreference.com/w/cpp/container/priority_queue.

The **priority_queue** operations include:

- Function **push** inserts an element at the appropriate location based on **priority order**.
- Function **pop** removes the **priority_queue**'s **highest-priority** element.
- Function **top** gets a reference to the **priority_queue**'s **top** element—implemented by calling the underlying container's **front** member function.

- Function **empty** determines whether the **priority_queue** is empty—implemented by calling the underlying container’s **empty** member function.
- Function **size** gets the **priority_queue**’s number of elements—implemented by calling the underlying container’s **size** member function.

Figure 13.12 demonstrates the **priority_queue** adaptor. Line 7 instantiates a **prior-ity_ que-ue** that stores double values and uses a **vector** as the underlying data structure. Lines 10–12 use member function **push** to add elements to the **priority_ que-ue**. Lines 17–20 use member function **empty** (available in all containers) to determine whether the **priority_ que-ue** is empty (line 17). If not, line 18 uses **priority_ que-ue** function **top** to retrieve the **highest-priority** element (i.e., the largest value) in the **priority_ que-ue** for output. Line 19 calls **pop**, available in all adaptor classes, to remove the **priority_ que-ue**’s **highest-priority** element.

[Click here to view code image](#)

```

1 // fig13_12.cpp
2 // Standard library priority_queue adaptor
3 #include <iostream>
4 #include <queue> // priority_queue adaptor
5
6 int main() {
7     std::priority_queue<double> priorities;
8
9     // push elements onto priorities
10    priorities.push(3.2);
11    priorities.push(9.8);
12    priorities.push(5.4);
13
14    std::cout << "Popping from priorities: "
15
16    // pop element from priority_queue

```

```
17     while (!priorities.empty()) {
18         std::cout << priorities.top() << ' ';
19         priorities.pop(); // remove top element
20     }
21
22     std::cout << "\n";
23 }
```

< >

Popping from priorities: 9.8 5.4 3.2

Fig. 13.12 | Standard library priority_queue adaptor class.

13.11 **bitset** Near Container

Class **bitset** makes it easy to create and manipulate **bit sets**, which are useful for representing a set of bit flags. bitsets are fixed in size at compile time. Class **bitset** is an alternate tool for bit manipulation, discussed in Appendix E.

The declaration

```
bitset<size> b;
```

creates **bitset** **b**, in which every one of the **size** bits is initially 0 (“off”).

The statement

```
b.set(bitNumber);
```

sets bit **bitNumber** of **bitset** **b** “on.” The expression **b.set()** sets all bits in **b** “on.”

The statement

```
b.reset(bitNumber);
```

sets bit **bitNumber** of **bitset** **b** “off.” The expression **b.reset()** sets all bits in **b** “off.”

The statement

```
b.flip(bitNumber);
```

“flips” bit `bitNumber` of `bitset b` (e.g., if the bit is “on”, `flip` sets it “off”). The expression `b.flip()` flips all bits in `b`.

The statement

```
b[bitNumber];
```

returns a reference to the `bool` at position `bitNumber` of `bitset b`. Similarly,

```
b.at(bitNumber);
```

performs range checking on `bitNumber` first. Then, if `bitNumber` is in range (based on the number of bits in the `bitset`), `at` returns a reference to the bit. Otherwise, `at` throws an `out_of_range` exception.

The statement

```
b.test(bitNumber);
```

performs range checking on `bitNumber` first. If `bitNumber` is in range (based on the number of bits in the `bitset`), `test` returns `true` if the bit is on, `false` if it’s off. Otherwise, `test` throws an `out_of_range` exception.

The expression

```
b.size()
```

returns the number of bits in `bitset b`.

The expression

```
b.count()
```

returns the number of bits that are set (`true`) in `bitset b`.

The expression

```
b.any()
```

returns `true` if any bit is set in `bitset b`.

The expression

```
b.all()
```

11 (added in C++11) returns `true` if all of the bits are set (`true`) in bitset `b`.

The expression

```
b.none()
```

returns `true` if none of the bits is set in bitset `b` (that is, all the bits are `false`).

The expressions

```
b == b1  
b != b1
```

compare the two bitsets for equality and inequality, respectively.

Each of the bitwise assignment operators `&=`, `|=` and `^=` (discussed in detail in Section E.5) can be used to combine bitsets. For example,

```
b &= b1;
```

performs a bit-by-bit logical AND between bitsets `b` and `b1`. The result is stored in `b`. Bitwise logical OR and bitwise logical XOR are performed by

```
b |= b1;  
b ^= b2;
```

The expression

```
b >>= n;
```

shifts the bits in bitset `b` right by `n` positions.

The expression

```
b <<= n;
```

shifts the bits in bitset `b` left by `n` positions.

The expressions

```
b.to_string()  
b.to_ulong()
```

convert `bitset b` to a string and an `unsigned long`, respectively.

13.12 Optional: A Brief Intro to Big O

In this chapter, you've seen Big *O* notations of $O(1)$, $O(n)$ and $O(\log n)$. In this section, we'll explain what those mean and introduce two others— $O(n^2)$ and $O(n \log n)$. All of these “Big *O*” expressions characterize the amount of work an algorithm must do when processing n items. On today's desktop computers, which may process a few billion operations-per-second, that translates to whether a program will run almost instantaneously or take seconds, minutes, hours, days, months, years or even more to complete. Obviously, you'd prefer algorithms that complete quickly, even though the number of items n that they may be processing is large. This is especially true in today's world of Big Data computing applications. In the implementation of the standard library containers and algorithms, Big *O*s of $O(1)$, $O(n)$, $O(\log n)$ and even $O(n \log n)$ —which is common for relatively good sorting algorithms—are considered to be reasonably efficient. Algorithms categorized by $O(n^2)$ or worse, such as $O(2^n)$ or $O(n!)$, could run on a modest number of items for centuries, millennia or longer. So you'll want to avoid writing such algorithms.

Searching algorithms all accomplish the same goal—finding in a container (which for this discussion we'll assume is an `array` container) an element matching a given search key, if such an element does, in fact, exist. There are, however, a number of things that differentiate search algorithms from one another. **The major difference is the amount of effort they require to complete the search, based on the number of items they must search through.** One way to describe this effort is with **Big *O* notation**, which indicates how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends particularly on how many data elements there are.

O(1) Algorithms

Suppose an algorithm is designed to test whether the first element of an `array` is equal to the second. If the `array` has 10 elements, this algorithm requires one comparison. If the `array` has 1000 elements, it still requires one comparison. In fact, the algorithm is completely independent of the number, n , of elements in the `array`. This algorithm is said to have **constant**

run time, which is represented in Big O notation as **$O(1)$** and pronounced as “order one.” An algorithm that’s **$O(1)$** does not necessarily require only one comparison. **$O(1)$** just means that the number of comparisons is *constant*—it does not grow as the size of the **array** increases. An algorithm that tests whether the first element of an **array** is equal to any of the next three elements is still **$O(1)$** even though it requires three comparisons.

$O(n)$ Algorithms

An algorithm that tests whether the first **array** element is equal to *any* of the other **array** elements will require at most $n - 1$ comparisons, where n is the number of **array** elements. If the **array** has 10 elements, this algorithm requires up to nine comparisons. If the **array** has 1000 elements, it requires up to 999 comparisons. As n grows larger, the n part of the expression $n - 1$ “dominates,” and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described earlier) is said to be **$O(n)$** . An **$O(n)$** algorithm is referred to as having a **linear run time**. **$O(n)$** is often pronounced “on the order of n ” or simply “order n .”

$O(n^2)$ Algorithms

Computer science students learn that, unfortunately, it is easy to write simple sorting algorithms that are $O(n^2)$. In Data Structures and Algorithms courses, the students learn that with clever algorithm design it’s possible to write sorting algorithms that are $O(n \log n)$ —these are far more efficient than $O(n^2)$ algorithms.

Efficiency of the Linear Search **$O(n)$**

The linear search algorithm, which is typically used for searching an unsorted **array**, runs in **$O(n)$** time. The worst case in this algorithm is that every element must be checked to determine whether the search key exists in the **array**. If the size of the **array** is doubled, the number of comparisons that the algorithm must perform is also doubled. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the **array**. But we seek algorithms that perform well, on average, across all searches, including those where the element

matching the search key is near the end of the **array**.

Linear search is easy to program, but it can be slow compared to other search algorithms, especially as n gets large. If a program needs to perform many searches on large **arrays**, it's better to implement a more efficient algorithm, such as the `binary_search` algorithm, which we'll use in [Chapter 14](#).

Perf  Sometimes the simplest algorithms perform poorly. Their virtue often is that they're easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.

Efficiency of the Binary Search $O(\log n)$

In the worst-case scenario, searching a *sorted array* of 1023 elements ($2^{10} - 1$) takes *only 10 comparisons* when using a binary search. The binary search algorithm compares the search key to the value in the middle of the sorted **array**. If it's a match, the search is over. More likely the search key will be larger or smaller than the middle element. If it's larger, we can eliminate the first half of the **array** from consideration. If it's smaller than the middle element we can eliminate the second half of the **array** from consideration. This creates a halving effect so that on subsequent searches, we have to search only 511, 255, 127, 63, 31, 15, 7, 3 and 1 elements. The number 1023 ($2^{10} - 1$) needs to be halved only 10 times to either find the key or determine that it's not in the **array**. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an **array** of 1,048,575 ($2^{20} - 1$) elements takes a *maximum of 20 comparisons* to find the key, and an **array** of about one billion elements takes a *maximum of 30 comparisons* to find the key. This is a tremendous improvement in performance over the linear search. For a one-billion-element **array**, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted **array** is the exponent of the first power of 2 greater than the number of elements in the **array**, which is represented as $\log_2 n$. All logarithms grow at "roughly the same rate," so for Big O comparison purposes the base can be omitted. This results in a Big O of **$O(\log n)$** for a binary search, which is also known as **logarithmic run time** and pronounced as "order log n ."

Common Big O Notations

The following table lists various common Big O notations along with a number of values for n to highlight the differences in the growth rates. If you interpret the values in the table as seconds of calculation, you can easily see why $O(n^2)$ algorithms are to be avoided!

$n =$	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	1	0	1	0	1
2	1	1	2	2	4
3	1	1	3	3	9
4	1	1	4	4	16
5	1	1	5	5	25
10	1	1	10	10	100
100	1	2	100	200	10,000
1000	1	3	1000	3000	10^6
1,000,000	1	6	1,000,000	6,000,000	10^{12}

13.13 Optional: A Brief Intro to Hash Tables

When a program creates objects, it may need to store and retrieve them efficiently. Storing and retrieving information with **arrays** is efficient if some aspect of your data directly matches a numerical key value and if the *keys are unique* and tightly packed. If you have 100 employees with nine-digit social security numbers and you want to store and retrieve employee data by using the social security number as an **array** index, the task will require an **array** with over 800 million elements, because nine-digit Social Security numbers must begin with 001–899 (excluding 666) as per the Social Security Administration's website:

[Click here to view code image](#)

<https://www.ssa.gov/employer/randomization.html>

This is impractical for most applications that use social security numbers as

keys. A program having so large an **array** could achieve high performance for both storing and retrieving employee records by simply using the social security number as the **array** index.

Numerous applications have this problem—namely, that either the keys are of the wrong type (e.g., not positive integers that correspond to **array** subscripts) or they’re of the right type, but *sparsely* spread over a *huge range*, such as social-security numbers for all the employees in a small company. What is needed is a high-speed scheme for converting keys such as social-security numbers, inventory part numbers and the like into unique **array** indices over a modest-size **array**. Then, when an application needs to store something, the scheme can convert the application’s key rapidly into an **array** index (a process called **hashing**), and the data can be stored at that slot in the **array**. Retrieval is accomplished the same way: Once the application has a key for which it wants to retrieve the data, the application simply applies the conversion to the key (again, called hashing)—this produces the **array** index where the data is stored and retrieved.

Why the name hashing? When we convert a key into an **array** index, we literally scramble the bits, forming a kind of “mishmashed,” or hashed, number. The number actually has no real significance beyond its usefulness in storing and retrieving particular data.

A glitch in the scheme is called a **collision**—this occurs when two different keys “hash into” the same cell (or element) in the **array**. We cannot store two values in the same space, so we need to find an alternative home for all values beyond the first that hash to the same **array** index. There are many schemes for doing this. One is to “hash again”—that is, to apply another hashing transformation to the previous hash result to provide the next candidate cell in the **array**. The hashing process is designed to distribute the values throughout the table, so hopefully an available cell will be found with one or a few hashes.

Another scheme uses one hash to locate the first candidate cell. If that cell is occupied, adjacent cells are searched sequentially until an available cell is found. Retrieval works the same way: The key is hashed once to determine the initial location and check whether it contains the desired data. If it does, the search is finished; otherwise, successive cells are searched sequentially until the desired data is found. A popular solution to hash-table collisions is

to have each cell of the table be a **hash bucket**—typically a linked list of all the key–value pairs that hash to that cell.

A hash table’s **load factor** affects the performance of hashing schemes. The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table. The closer this ratio gets to 1.0, the greater the chance of collisions, which slow data insertion and retrieval.

Perf  The load factor in a hash table is a classic example of a memory-space/execution-time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

C++’s **unordered_set**, **unordered_multiset**, **unordered_map** and **unordered_multimap** associative containers are implemented as hash tables “under the hood.” When you use those containers, you get the benefit of high-speed data storage and retrieval without having to build your own hash-table mechanisms—a classic example of reuse. This concept is profoundly important in our study of object-oriented programming. As discussed in earlier chapters, classes encapsulate and hide complexity (i.e., implementation details) and offer user-friendly interfaces. Properly crafting classes to exhibit such behavior is one of the most valued skills in the field of object-oriented programming.

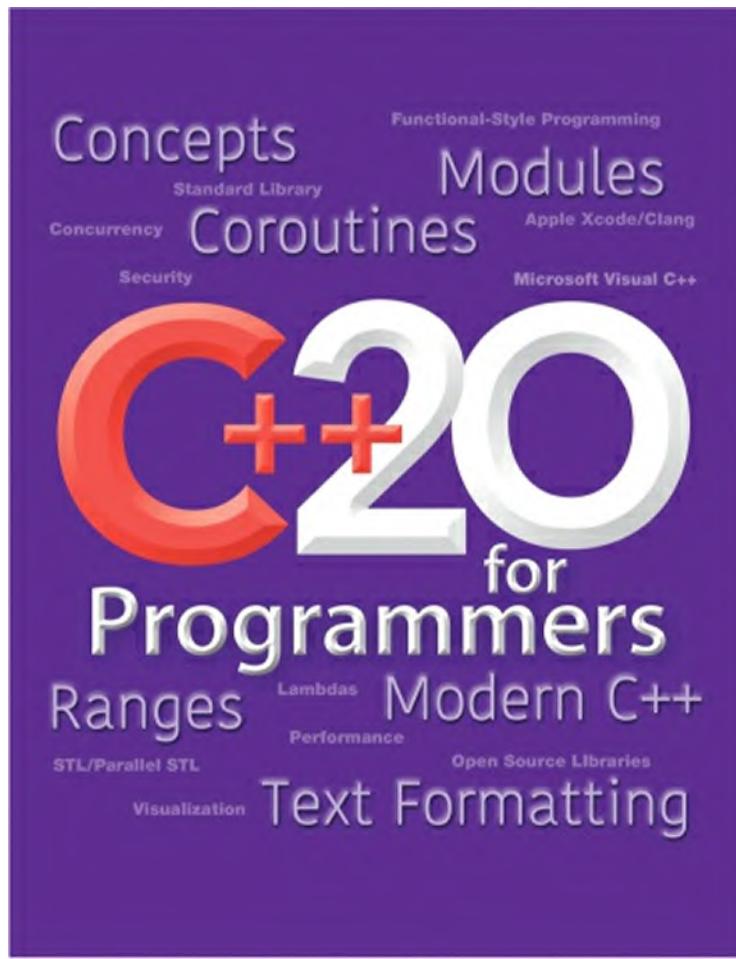
13.14 Wrap-Up

In this chapter, we introduced three key components of the standard library—containers, iterators and algorithms. You learned about the linear **sequence containers**, **array** (Chapter 6), **vector**, **deque**, **forward_list** and **list**, which all represent linear data structures. We discussed the nonlinear **associative containers**, **set**, **multiset**, **map** and **multi-map** and their unordered versions. You also saw that the **container adaptors** **stack**, **queue** and **priority_queue** can be used to restrict the operations of the sequence containers **vector**, **deque** and **list** for the purpose of implementing the specialized data structures represented by the container adaptors. You learned the categories of iterators and that each algorithm can

be used with any container that supports the minimum iterator functionality the algorithm requires. We distinguished between common ranges and C++20 ranges and demonstrated the `std::ranges::copy` algorithm. You also learned the features of class `bitset`, which makes it easy to create and manipulate bit sets as a container.

[Chapter 14](#) continues our discussion of the standard library's containers, iterators and algorithms with a detailed treatment of algorithms. You'll see that C++20 concepts have been used extensively in the latest version of the standard library. We'll discuss the minimum iterator requirements that determine which containers can be used with each algorithm. We'll continue our discussion of lambda expressions and see that function pointers and function objects (instances of classes that overload the parentheses operator) can be passed to algorithms. In [Chapter 15](#), you'll see how C++20 concepts come into play when we build a custom container and a custom iterator.

Chapter 14. Standard Library Algorithms and C++20 Ranges & Views



Objectives

In this chapter, you'll:

- Understand minimum iterator requirements for working with standard library containers and algorithms.
- Create lambda expressions that capture local variables to use in the lambda expressions' bodies.
- Use many of the C++20 `std::ranges` algorithms.

- Understand the C++20 concepts that correspond to the C++20 `std::ranges` algorithms' minimum iterator requirements.
 - Compare the new C++20 `std::ranges` algorithms with older common-range `std` algorithms.
 - Use iterators with algorithms to access and manipulate the elements of standard library containers.
 - Pass lambdas, function pointers and function objects into standard library algorithms mostly interchangeably.
 - Use projections to transform objects in a range before processing them with C++20 range algorithms.
 - Use C++20 views and lazy evaluation with C++20 ranges.
 - Learn about C++ ranges features possibly coming in C++23.
 - Be introduced to parallel algorithms for performance enhancement —we'll discuss these in [Chapter 16](#).
-

Outline

[14.1 Introduction](#)

[14.2 Algorithm Requirements: C++20 Concepts](#)

[14.3 Lambdas and Algorithms](#)

[14.4 Algorithms \(Mostly C++20 `std::ranges` Versions\)](#)

[14.4.1 `fill`, `fill_n`, `generate` and `generate_n`](#)

[14.4.2 `equal`, `mismatch` and
`lexicographical_compare`](#)

[14.4.3 `remove`, `remove_if`, `remove_copy` and
`remove_copy_if`](#)

[14.4.4 `replace`, `replace_if`, `replace_copy` and
`replace_copy_if`](#)

[14.4.5 Mathematical Algorithms](#)

[14.4.6 Searching and Sorting Algorithms](#)

[14.4.7 `swap`, `iter_swap` and `swap_ranges`](#)

[14.4.8 `copy_backward`, `merge`, `unique`, `reverse`,](#)

- copy_if and copy_n
 - 14.4.9 inplace_merge, unique_copy and reverse_copy
 - 14.4.10 Set Operations
 - 14.4.11 lower_bound, upper_bound and equal_range
 - 14.4.12 min, max and minmax
 - 14.4.13 Algorithms gcd, lcm, iota, reduce and partial_sum from Header <numeric>
 - 14.4.14 Heapsort
- 14.5 Function Objects (Functors)
- 14.6 Projections
- 14.7 C++20 Views and Functional-Style Programming
 - 14.7.1 Range Adaptors
 - 14.7.2 Working with Range Adaptors and Views
- 14.8 Intro to Parallel Algorithms
- 14.9 Standard Library Algorithm Summary
- 14.10 A Look Ahead to C++23 Ranges
- 14.11 Wrap-Up
-

14.1 Introduction

This chapter discusses the standard library's algorithms, focusing on common container manipulations, including **filling with values**, **generating values**, **comparing elements or entire containers**, **removing elements**, **replacing elements**, **mathematical operations**, **searching**, **sorting**, **swapping**, **copying**, **merging**, **set operations**, **determining boundaries**, and **calculating minimums and maximums**. The standard library provides many pre-packaged, templatized algorithms:

- 20 • 90 in the `<algorithm>` header's `std` namespace—82 also are overloaded in the `std::ranges` namespace for use with **C++20 ranges**,
- 11 in the `<numeric>` header's `std` namespace—none are overloaded in the **C++20 std::ranges namespace**, and

20 • 14 in the `<memory>` header's `std` namespace—all 14 also are overloaded in the `std::ranges` namespace for use with **C++20 ranges**.

11 17 20 Many algorithms were added in C++11 and C++20, and a few in C++17. The `<algorithm>` header contains **mutating sequence algorithms**, **nonmodifying sequence algorithms** and **sorting and related algorithms**. Many have parallel versions (which we overview in [Section 14.8](#) and demonstrate in [Chapter 17](#), Concurrency, Parallelism, C++20 Coroutines and the Parallel STL). For the complete list of algorithms and their descriptions, visit:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/algorithm>

20 Minimum Algorithm Requirements and C++20 Concepts

Concepts  The standard library's algorithms specify **minimum requirements** that help you determine which **containers**, **iterators** and **functions** can be passed to each algorithm. The **C++20 range-based algorithms** in the `std::ranges` namespace use **C++20 concepts** feature to specify their requirements. We briefly introduce **C++20 concepts** as needed for you to understand the requirements for working with these algorithms. We'll discuss concepts in more depth in [Chapter 15](#) as we build templates.

C++20 Range-Based Algorithms vs. Earlier Common-Range Algorithms

Most of the algorithms we present in this chapter have overloads in

- 20** • the `std::ranges` namespace for use with **C++20 ranges**, and
• the `std` namespace for use with **pre-C++20 common ranges**.

We'll focus mainly on the **C++20 ranges versions**.^{1,2}

1. Tristan Brindle, “An Overview of Standard Ranges,” September 29, 2019. Accessed May 1, 2021.
<https://www.youtube.com/watch?v=SYLgg7Q5Zws>.
2. Tristan Brindle, “C++20 Ranges in Practice,” October 8, 2020. Accessed May 1, 2021.
https://www.youtube.com/watch?v=d_E-VLyUnzc.

Lambdas, Function Pointers and Function Objects

Section 6.14.2 introduced **lambda expressions**. In Section 14.3, we'll revisit them and introduce additional details. As you'll see, various algorithms can receive a **lambda**, a **function pointer** or a **function object** as an argument. Most examples in this chapter use **lambdas** because they're convenient for expressing small tasks. In Section 14.5, you'll see that a **lambda expression** often is interchangeable with a **function pointer** or a **function object** (also called a **functor**). A function object's class overloads the **operator() function**, allowing the object's name to be used as a function name, as in *objectName(arguments)*. **Lambdas** that use variables from their enclosing scope are converted to function objects by the compiler.

C++20 Views and Functional-Style Programming with Lazy Evaluation

Like many modern languages, C++ offers **functional-style programming** capabilities, which we began introducing in Section 6.14.3. In particular, we demonstrated **functional-style filter, map and reduce operations**. In Section 14.7, we'll continue our presentation of functional-style programming with C++20's new **<ranges> library**.

Parallel Algorithms

C++17 introduced new parallel overloads for 69 standard library algorithms in the header **<algorithm>**, enabling you to take advantage of **multi-core architectures** to enhance program performance. Section 14.8 briefly overviews the parallel overloads. We enumerate the algorithms with parallel versions in Section 14.9's standard-library-algorithms summary tables. Chapter 16 demonstrates several parallel algorithms. In that chapter, we'll use features from the **<chrono>** header to time standard library algorithms running sequentially on a single core and running parallel on multiple cores so you can see the performance improvement.

Looking Ahead to Ranges in C++23

Some C++20 algorithms—including those in the **<numeric> header** and the parallel algorithms in the **<algorithm> header**—do not have `std::ranges` overloads. Section 14.7 overviews updates expected in C++23 and mentions the **open-source project ranges-next**,³ which contains implementations for some of the proposed updates.

3. Corentin Jabot, “Ranges For C++23.” Accessed April 30, 2021.
<https://github.com/cor3ntin/rangesnext>.

20 14.2 Algorithm Requirements: C++20 Concepts

SE A The C++ standard library separates containers from the algorithms that manipulate the containers. Most algorithms operate on container elements indirectly via iterators. **This architecture makes it easier to write generic algorithms applicable to a variety of containers.** This is a strength of the STL.

Container class templates and their corresponding iterator class templates typically reside in the same header. For example, the `<array>` header contains the templates for class `array` and its iterator class. A container internally creates objects of its iterator class and returns them via container member functions, such as `begin`, `end`, `cbegin` and `cend`.

Iterator Requirements

SE A For maximum reuse, **each algorithm can operate portably on any container that meets the algorithm's minimum iterator requirements.**⁴ So, an algorithm that requires **forward iterators** can operate on any container that provides **at least forward iterators**.

4. Alexander Stepanov and Meng Lee, “The Standard Template Library, Section 2: Structure of the Library,” October 31, 1995. Accessed May 10, 2021.
<http://stepanovpapers.com/STL/DOC.PDF>.

The `vector` and `array` containers support **random-access iterators**. These provide every iterator operation discussed in [Section 13.3](#). This means **all standard library algorithms can operate on vectors**, and **all algorithms that do not modify a container's size can operate on arrays**.

Before C++20,

- each container's documentation mentioned the iterator level it supported, and
- each algorithm's documentation mentioned its minimum iterator requirements.

Programmers were expected to adhere to an algorithm's documented

requirements by using only containers with iterators that satisfied those requirements. The compiler could not prevent you from passing incorrect iterators to an algorithm, so it was easy to pass the wrong kinds of iterators. The compiler would then substitute those incorrect iterators' types throughout the algorithm's template definition. Bjarne Stroustrup observed that this led to "spectacularly bad error messages."⁵

5. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—1. A bit of background," January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.

20 C++20 Concepts

One of the "big four" C++20 features is **concepts**—a technology for restricting the types used with templates. Stroustrup points out that "Concepts complete C++ templates as originally envisioned"⁶ decades ago. Each concept specifies a type's requirements or a relationship between types.⁷

6. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—Conclusion," January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.

7. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—3.1 Specifying template interfaces," January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.

When a concept is applied to an algorithm's parameter, the compiler can check the requirements in the algorithm's call before substituting the argument's type throughout the function template. If your argument's type does not satisfy the concept's requirements, a benefit of concepts is that the compiler produces many fewer and much clearer error messages than for the older **common-range algorithms**. This makes it easier for you to understand the errors and correct your code.

20 This chapter's primary focus is **C++20's new range-based algorithms**, which are **constrained with many C++20 predefined concepts**. There are 76 predefined concepts^{8,9} in the standard. Though we've used standard library templates extensively in [Chapters 6](#) and [13](#) and will do so again here, we have not used concepts in the code, nor will we in this chapter. The programmers responsible for creating **C++20's range-based algorithms** used concepts in the algorithms' prototypes to specify the algorithms' iterator and range requirements.

8. "Index of library concepts." Accessed May 10, 2021. <https://eel.is/c++draft/conceptindex>.

9. The standard also specifies 30 “exposition-only” concepts which are used for discussion purposes.
“Exposition-only in the C++ standard?” Answered December 28, 2015. Accessed May 10, 2021.
<https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard>.

Concepts When invoking **C++20’s range-based algorithms**, you must pass container and iterator arguments that meet the algorithms’ requirements. So, for the algorithms we present in this chapter, we’ll mention the predefined concept names specified in the algorithms’ prototypes, and we’ll briefly explain how they constrain the algorithms’ arguments. We’ll call out the concepts with the icon you see in the margin next to this paragraph. In [Chapter 15](#), we’ll take a template developer’s viewpoint as we demonstrate implementing custom templates with concepts.

C++20 Iterator Concepts

20 Concepts As you view the **C++20 range-based algorithms**’ documentation, you’ll often see in their prototypes the following **C++20 iterator concepts**, which are defined in namespace `std` in the `<iterator>` header:

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

These specify the type requirements for the **iterator categories** we introduced in [Chapter 13](#). As we present constrained algorithms here in [Chapter 14](#), we’ll briefly introduce these concepts and others that place additional restrictions on **iterators**, including

- `indirectly_copyable`,
- `indirectly_readable`,
- `indirectly_writable` and
- `weakly_incrementable`.

For a complete list of **iterator concepts**, see the “C++20 iterator concepts” section at:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/iterator>

20 C++20 Range Concepts

Concepts  The **range concept** describes a type with a **begin iterator** and an **end sentinel**, possibly of different types. You’ll often see the following **C++20 ranges concepts** in the **C++20 range-based algorithms**’ prototypes:

- **input_range**—a range that supports **input_iterators**.
- **output_range**—a range that supports **output_iterators**.
- **forward_range**—a range that supports **forward_iterators**.
- **bidirectional_range**—a range that supports **bidirectional_iterators**.
- **random_access_range**—a range that supports **random_access_iterators**.
- **contiguous_range**—a range that supports **contiguous_iterators**.

These are defined in the **std::ranges** namespace in the **<ranges>** header. They specify the requirements for **ranges** supporting the **iterator categories** discussed in [Chapter 13](#). We’ll discuss other concepts as we encounter them in the coming chapters.

14.3 Lambdas and Algorithms

You can customize the behavior of many standard library algorithms by passing a function as an argument. You saw in [Section 6.14.2](#) that **lambda expressions define anonymous functions** inside other functions and that they can manipulate the enclosing function’s local variables. Throughout this chapter, we’ll typically pass **lambdas** to standard library algorithms because they’re convenient for expressing small tasks.

Figure 14.1 revisits the standard library's **copy** and **for_each** algorithms, this time using the versions from C++20's **std::ranges** namespace. Recall that **for_each** receives as one of its arguments a function specifying a task to perform on each container element.

[Click here to view code image](#)

```
1 // fig14_01.cpp
2 // Lambda expressions.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{1, 2, 3, 4}; // initial values
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14}
```

< >

```
values contains: 1 2 3 4
```

Fig. 14.1 | Lambda expressions.

Algorithm **copy** and Common Ranges Iterator Requirements vs. C++20 Ranges Iterator Requirements

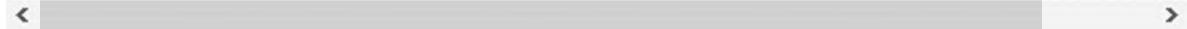
20 Line 9 creates an array of **int** values, which line 13 displays using an **ostream_iterator** and the **copy algorithm**¹⁰ from C++20's **std::ranges** namespace. Section 13.6.2 called the **common ranges std::copy** algorithm as follows:

10. “`std::ranges::copy`, `std::ranges::copy_if`, `std::ranges::copy_result`, `std::ranges::copy_if_result`.” Accessed April 28, 2021.

<https://en.cppreference.com/w/cpp/algorithm/ranges/copy>.

[Click here to view code image](#)

```
std::copy(integers.cbegin(), integers.cend(), out)
```



This algorithm's documentation states that the first two arguments must be **input iterators** designating the beginning and end of the **common range** to copy. The third argument was an **output iterator** indicating where to copy the elements.

Line 13 calls the `std::ranges` version of `copy` with just two arguments—the values container and an **ostream_iterator**. This version of `copy` determines values' beginning and end for you by calling

```
std::ranges::begin(values)
```

and

```
std::ranges::end(values)
```

Concepts Any container that supports `begin` and `end` iterators can be treated as a C++20 range. The first argument to the `std::ranges::copy` algorithm must be an `input_range`. The second must be a `weakly_incrementable output iterator`—that is, it supports the `++ operator` for moving to the next element and can be used to output elements. Here, we write to the standard output stream, but we also could write into another `range`.

SE A 20 Err Use C++20's range-based algorithms rather than the older common-ranges algorithms. Passing an entire container, rather than `begin` and `end` iterators, simplifies your code and eliminates accidentally mismatched iterators—that is, a `begin` iterator that points to one container and an `end` iterator that points to a different container.

Algorithm `for_each`

20 This example uses the `for_each` algorithm from C++20's `std::ranges` namespace twice. The first call in line 17 multiples each `values` element by 2 and displays the result:

[Click here to view code image](#)

```
15 // output each element multiplied by two
16 std::cout << "\nDisplay each element multip
17 std::ranges::for_each(values, [](auto i) { s
18
```

```
< >
Display each element multiplied by two: 2 4 6 8
```

Section 11.6.13 called the **common ranges `std::for_each` algorithm** as follows:

[Click here to view code image](#)

```
for_each(begin(items), end(items), [](auto& item)
```

In that call, the first two arguments are **input iterators** designating the beginning and end of a **common range** of elements to process. The third is a **lambda** specifying the task to perform on each element.

Line 17 calls the **`std::ranges::for_each` algorithm**, which has **two arguments**:

Concepts • an **input_range** (`values`) containing the elements to copy, and

Concepts • a function with one argument, which is allowed, but not required, to modify its argument in the range. The **for_each algorithm** describes this parameter with the concept **indirectly_unary_invocable**, meaning a **function that accesses a value indirectly by dereferencing the function's iterator argument**.

The **for_each algorithm** repeatedly calls the function in its second argument, passing one element at a time from its **input_range** argument. For a complete list of **indirect callable concepts** like

indirectly_unary_invocable, see the “Algorithm concepts and utilities” section at:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/iterator>

Lambda with an Empty Introducer

The **lambda** in line 17 multiplies its parameter *i* by 2 and displays the result. Recall that **lambdas** begin with the **lambda introducer** (`[]`), followed by a parameter list and function body. **This lambda introducer is empty, so it does not capture any of main's local variables.** The parameter's type is `auto`, so this is a generic lambda for which the compiler infers the type, based on the context. Since `values` contains `ints`, the compiler infers parameter *i*'s type as `int`.

The line 17 lambda is similar to the standalone function

```
void timesTwo(int i) {  
    cout << i * 2 << " "  
}
```

If we defined this function, we could have passed it to **for_each**, as in

[Click here to view code image](#)

```
std::ranges::for_each(values, timesTwo);
```

Lambda with a Nonempty Introducer—Capturing Local Variables

Line 21 calls **for_each** to total the elements of `values`. The **lambda introducer** [`&sum`] in

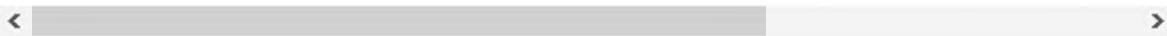
```
[&sum] (auto i) {sum += i;}
```

 captures¹¹ the local variable `sum` (defined in line 20) by reference¹² (`&`), so the lambda can modify `sum`'s value. The **for_each algorithm** passes each element of `values` to the lambda, which adds the element to the `sum`. Line 22 displays the `sum`. **Without the ampersand, sum would be captured by value,**¹³ so lambda would not modify the local variable in the enclosing function's scope.

11. “F.50: Use a lambda when a function won’t do (to capture local variables, or to write a local function).” Accessed April 30, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-capture-vs-overload>.
12. “F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms.” Accessed April 30, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-reference-capture>.
13. “F.53: Avoid capturing by reference in lambdas that will be used non-locally, including returned, stored on the heap, or passed to another thread.” Accessed April 30, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-value-capture>.

[Click here to view code image](#)

```
19     // add each element to sum
20     int sum{0}; // initialize sum to zero
21     std::ranges::for_each(values, [&sum] (auto& value) {
22         sum += value;
23     })
```



Sum of values

Lambda Introducers [&] and [=]

A **lambda introducer** can capture multiple variables from the enclosing function’s scope by providing a comma-separated list of variables. You also may capture multiple variables using **lambda introducers** of the form `[&]` or `[=]`:

- The **lambda introducer** `[&]` indicates that every variable from the enclosing scope used in the **lambda’s body** should be captured **by reference**.
- The **lambda introducer** `[=]` indicates that every variable from the enclosing scope used in the **lambda’s body** should be captured **by value**.

Lambda Return Types

The compiler can infer a **lambda's return type** if the body contains a statement of the form

```
return expression;
```

Otherwise, the **lambda's return type** is `void` unless you explicitly specify a return type using C++11's **trailing return type** syntax (`-> type`), as in

```
[] (parameterList) -> type {lambdaBody}
```

The trailing return type is placed between the parameter list's closing right parenthesis and the lambda's body.

14.4 Algorithms¹⁴

[14.](#) Most of the algorithms we present are the new C++20 `std::ranges` versions.

Sections [14.4.1](#)–[14.4.12](#) demonstrate many of the standard library algorithms.

14.4.1 `fill`, `fill_n`, `generate` and `generate_n`

[20](#) Figure 14.2 demonstrates algorithms `fill`, `fill_n`, `generate` and `generate_n` from C++20's `std::ranges` namespace:

- Algorithms `fill` and `fill_n` set every element in a range of container elements to a specific value.
- Algorithms `generate` and `generate_n` use a **generator function** to create values for every element in a range of container elements. The generator function takes no arguments and returns a value.

Lines 9–12 define a generator function `nextLetter` as a standalone function. We'll also implement this as a lambda, so you can see the similarities. Line 15 defines a 10-element `char` array `chars` that we'll manipulate in this example.

[Click here to view code image](#)

```
1 // fig14_02.cpp
2 // Algorithms fill, fill_n, generate and ge:
3 #include <algorithm> // algorithm definitio:
```

```

4  #include <array> // array class-template de
5  #include <iostream>
6  #include <iterator> // ostream_iterator
7
8  // generator function returns next letter (
9  char nextLetter() {
10     static char letter{'A'};
11     return letter++;
12 }
13
14 int main() {
15     std::array<char, 10> chars{};


```



Fig. 14.2 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`.

fill Algorithm

20 Concepts  Line 16 uses the **C++20 `std::ranges::fill algorithm`** to place the character in every `chars` element. The first argument must be an **output_range** so that the algorithm can

- iterate from the beginning to the end of the **range** by incrementing its **iterator** with `++`, and
- **dereference the iterator** to write a new value into the current element.

Concepts  An array has **contiguous_iterators**, which support all **iterator** operations. So, **fill** can increment the iterators with `++`. Also, the array `chars` is non-const, so **fill** can **dereference the iterators** to write values into the **range**. Line 20 displays `chars`' elements. We used bold text in the outputs to highlight the changes made to `chars` by each algorithm.

[Click here to view code image](#)

```

16     std::ranges::fill(chars, '5'); // fill char
17


```

```
18     std::cout << "chars after filling with 5s:  
19     std::ostream_iterator<char> output{std::cou  
20     std::ranges::copy(chars, output);  
21
```

```
< >  
chars after filling with 5s: 5 5 5 5 5 5 5 5 5 5
```

```
< >
```

[fill_n Algorithm](#)

20 Concepts Line 23 uses the **C++20 std::ranges:: fill_n algorithm** to place the character (the third argument) in chars' first five elements (specified by the second argument). The first argument must be at least an **output_iterator**. arrays support **contiguous_iterators** and chars is non-const, so calling chars.begin() returns an **iterator** that can be used to write values into the **range**.

[Click here to view code image](#)

```
22 // fill first five elements of chars with '  
23 std::ranges::fill_n(chars.begin(), 5, 'A');  
24  
25 std::cout << "\nchars after filling five el  
26 std::ranges::copy(chars, output);  
27
```

```
< >  
chars after filling five elements with A A A A A
```

```
< >
```

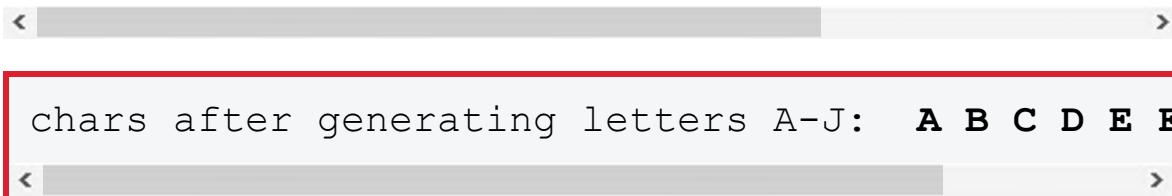
[generate Algorithm](#)

20 Concepts Line 29 uses the **C++20 std::ranges::**

generate algorithm to place the result of a call to **generator function** nextLetter in every element of chars. The first argument must be an **output_range**. Function nextLetter (lines 9–12) defines a static local char variable letter and initializes it to . Line 11 returns the current letter value, postincrementing it for use in the next call to the function.

[Click here to view code image](#)

```
28 // generate values for all elements of char
29 std::ranges::generate(chars, nextLetter);
30
31 std::cout << "\nchars after generating lett";
32 std::ranges::copy(chars, output);
33
```



```
< >
chars after generating letters A-J: A B C D E E
< >
```

[generate_n Algorithm](#)

20 Concepts Line 35 uses the **C++20 std::ranges::generate_n algorithm** to place the result of each call to **generator function** nextLetter in five elements of chars, starting from chars.begin(). The first argument must be an **input_or_output_iterator**, which is an **iterator** that can be incremented with ++ and **dereferenced with ***. All iterators satisfy this requirement. The first argument's **iterator** also must be **indirectly_writable**—when the algorithm **dereferences the iterator**, it must be able to write a new value into the **dereferenced element** in the range.

[Click here to view code image](#)

```
34 // generate values for first five elements
```

```
35     std::ranges::generate_n(chars.begin(), 5, n);
36
37     std::cout << "\nchars after generating K-O";
38     std::ranges::copy(chars, output);
39
```

```
chars after generating K-O into elements 0-4: K
```

Using the `generate_n` Algorithm with a Lambda

20 Lines 41–46 once again use the **C++20 `std::ranges::generate_n` algorithm** to place the result of a **generator function** call into `chars` elements, starting from `chars.begin()`. In this case, the **generator function** is implemented as a lambda (lines 42–45) with no arguments—as specified by the empty parentheses—that returns a generated letter. For a lambda with no arguments, the parameter list’s parentheses are not required. The compiler infers from the return statement that the **lambda’s return type** is `char`.

[Click here to view code image](#)

```
40     // generate values for first three elements
41     std::ranges::generate_n(chars.begin(), 3
42         []() { // lambda that takes no argument
43             static char letter{'A'};
44             return letter++;
45         }
46     );
47
48     std::cout << "\nchars after generating A";
49     std::ranges::copy(chars, output);
50     std::cout << "\n";
51 }
```

```
chars after generating A-C into elements 0-2: A
```

< >

14.4.2 **equal**, **mismatch** and **lexicographical_compare**

20 Figure 14.3 demonstrates comparing sequences of values for equality using algorithms **equal**, **mismatch** and **lexicographical_compare** from C++20's **std::ranges** namespace. Lines 12–14 create and initialize three arrays, then lines 17–22 display their contents.

[Click here to view code image](#)

```
1 // fig14_03.cpp
2 // Algorithms equal, mismatch and lexicogra...
3 #include <algorithm> // algorithm definitio...
4 #include <array> // array class-template de...
5 #include <fmt/format.h> // C++20: This will ...
6 #include <iomanip>
7 #include <iostream>
8 #include <iterator> // ostream_iterator
9 #include <string>
10
11 int main() {
12     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9,
13     std::array a2{a1}; // initializes a2 with ...
14     std::array a3{1, 2, 3, 4, 1000, 6, 7, 8,
15     std::ostream_iterator<int> output{std::cout,
16
17     std::cout << "a1 contains: ";
18     std::ranges::copy(a1, output);
19     std::cout << "\na2 contains: ";
20     std::ranges::copy(a2, output);
21     std::cout << "\na3 contains: ";
22     std::ranges::copy(a3, output);
23 }
```

< >

```
a1 contains: 1 2 3 4 5 6 7 8 9 10  
a2 contains: 1 2 3 4 5 6 7 8 9 10  
a3 contains: 1 2 3 4 1000 6 7 8 9 10
```

Fig. 14.3 | Algorithms `equal`, `mismatch` and `lexicographical_compare`.

`equal` Algorithm

20 Concepts Lines 26 and 30 use the **C++20 `std::ranges::equal` algorithm** to compare two `input_ranges` for equality. The algorithm returns `false` if the sequences are not the same length. Otherwise, it compares each `range`'s corresponding elements with the `==` operator, returning `true` if they're all equal and `false` otherwise. Line 26 compares the elements in `a1` to the elements in `a2`. In this example, `a1` and `a2` are equal. Line 30 compares `a1` and `a3`, which are not equal.

[Click here to view code image](#)

```
24 // compare a1 and a2 for equality  
25 std::cout << fmt::format("\n\na1 is equal to a2: {}\n",  
26 std::ranges::equal(a1, a2))  
27  
28 // compare a1 and a3 for equality  
29 std::cout << fmt::format("a1 is equal to a3: {}\n",  
30 std::ranges::equal(a1, a3))  
31
```

```
a1 is equal to a2: true  
a1 is equal to a3: false
```

`equal` Algorithm with Binary Predicate Function

Many standard library algorithms that compare elements enable you to pass a function that customizes how elements are compared. For example, you can pass to the **`equal`** algorithm a function that receives as arguments the two elements `equal` is comparing and returns a `bool` value telling `equal` whether those elements are equal. Because the function receives two arguments and returns a `bool`, it's referred to as a **binary predicate function**. This can be useful in ranges containing objects for which an `==` operator is not defined or containing pointers. For example, you can compare `Employee` objects for age, ID number, or location rather than comparing entire objects. You can compare what pointers refer to rather than comparing the addresses stored in the pointers.

20 The C++20 `std::ranges` algorithms also support **projections**, which enable algorithms to process subsets of each object in a **range**. For example, to sort `Employee` objects by their salaries, you can use a **projection** that selects each `Employee`'s salary. While sorting the `Employees`, they'd be compared only by their salaries to determine sort order and the `Employee` objects would be arranged accordingly. We'll perform this sort in [Section 14.6](#).

mismatch Algorithm

20 The C++20 `std::ranges:: mismatch algorithm` (line 33) compares two **input_ranges**. The algorithm returns a `std::ranges::mismatch_result`, which contains **iterators** named `in1` and `in2`, pointing to the mismatched elements in each **range**. If all the elements match, the `in1` is equal to the first **range**'s **sentinel**, and `in2` is equal to the second **range**'s **sentinel**. We infer the variable `location`'s type with `auto` (line 33). Line 35 determines the index of the mismatch with the expression

```
location.in1 - a1.begin()
```

which evaluates to the number of elements between the **iterators**—this is analogous to pointer arithmetic ([Chapter 7](#)). Like `equal`, `mismatch` can receive a **binary predicate function** to customize the comparisons ([Section 14.6](#)).

[Click here to view code image](#)

```
32 // check for mismatch between a1 and a3
33 auto location{std::ranges::mismatch(a1, a3)
34 std::cout << fmt::format("a1 and a3 mismatch at index {}",
35                         (location.in1 - a1.begin()))
36                         *location.in1, *location.in2);
37
```



```
a1 and a3 mismatch at index 4 (5 vs. 1000)
```

A Note Regarding `auto` and Algorithm Return Types

Template type declarations can become complex and error-prone quickly, as is commonly the case for standard library **algorithm** return types. Throughout this chapter, when we initialize a variable with an **algorithm**'s return value (as in line 33), we'll use `auto` to infer the variable's type. To help you understand why, consider the return type for the `std::ranges::mismatch` algorithm used in line 33:

[Click here to view code image](#)

```
std::ranges::mismatch_result<borrowed_iterator_t<R1>,
                           borrowed_iterator_t<R2>>
```



R1 and R2 are the types of the **ranges** passed to `mismatch`. Based on the declarations of `a1` and `a3`, the algorithm's return type in line 33 is

[Click here to view code image](#)

```
std::ranges::mismatch_result<borrowed_iterator_t<array<int, 10>>,
                           borrowed_iterator_t<array<int, 10>>>
```



As you can see, it's much more convenient to simply say `auto` and let the compiler determine this complex declaration for you.

lexicographical_compare Algorithm

20 Concepts C The C++20 std::ranges::

lexicographical_compare algorithm (lines 42–43) compares the contents of two **input_ranges**—in this case, strings. Like containers, **strings have iterators that enable them to be treated as C++20 ranges**. While iterating through the **ranges**, if there is a mismatch between their corresponding elements and the element in the first **range** is **less than** the corresponding element in the second, the algorithm returns **true**. Otherwise, the algorithm returns **false**. This algorithm can be used to arrange sequences lexicographically. It also can receive a **binary predicate function** that returns **true** if its first argument is less than its second.

[Click here to view code image](#)

```
38     std::string s1{"HELLO"};
39     std::string s2{"BYE BYE"};
40
41     // perform lexicographical comparison of c1
42     std::cout << fmt::format("{} < {}":           std::ranges::lexicographica
43                           std::ranges::lexicographica
44 }
```

< >

```
"HELLO" < "BYE BYE": false
```

14.4.3 remove, remove_if, remove_copy and remove_copy_if

Figure 14.4¹⁵ demonstrates removing values from a sequence with algorithms **remove**, **remove_if**, **remove_copy** and **remove_copy_if** from C++20's **std::ranges** namespace. Line 9 creates a `vector<int>` that we'll use to initialize other vectors in this example.

¹⁵. As of May 2021, `std::ranges::remove` did not compile on the most recent clang++ compiler.

[Click here to view code image](#)

```
1 // fig14_04.cpp
2 // Algorithms remove, remove_if, remove_copy
3 #include <algorithm> // algorithm definitio
4 #include <iostream>
5 #include <iterator> // ostream_iterator
6 #include <vector>
7
8 int main() {
9     std::vector init{10, 2, 15, 4, 10, 6};
10    std::ostream_iterator<int> output{std::cout,
11
```



Fig. 14.4 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`.

remove Algorithm

20 Concepts  Lines 12–14 create initialize the vector `v1` with a copy of `init`'s elements, then output `v1`'s contents. Line 17 uses the **C++20** `std::ranges:: remove algorithm` to eliminate from `v1` all elements with the value 10. The first argument must be a `forward_range` —a range that supports `forward_iterators`. A vector supports more powerful `random_access_iterators`, so a vector can work with this algorithm and any other algorithm that requires lesser iterators. This algorithm does not modify the container's number of elements and does not destroy the eliminated elements. Instead, it places at the beginning of the container all elements that are not eliminated. It returns an `iterator/sentinel pair` representing the subrange of container's elements that are no longer valid. **Those elements should not be used, so they're commonly erased from the container**, as we'll discuss momentarily.

[Click here to view code image](#)

```
12 std::vector v1{init}; // initialize with co
13 std::cout << "v1: ";
```

```
14     std::ranges::copy(v1, output);
15
16     // remove all 10s from v1
17     const auto& [begin1, end1]{std::ranges::remove_if(v1, v1.end(), []{return *it == 10;})};
18     v1.erase(begin1, end1);
19     std::cout << "\nv1 after removing 10s: ";
20     std::ranges::copy(v1, output);
21
```

```
v1: 10 2 15 4 10 6
v1 after removing 10s: 2 15 4 6
```

17 C++17 Structured Bindings

Line 17 uses the **iterator/sentinel pair** returned by `remove_if` to initialize the reference (&) variables `begin1` and `end1` using a C++17 **structured binding declaration**.^{16,17} This is sometimes referred to as **unpacking the elements** and can be used to extract into individual variables the elements of a built-in array, array object or tuple. Here, the **iterator** is assigned to `begin1` and the **sentinel** to `end1`. **Structured bindings** also can be used to unpack the public data members of a struct or class object and the elements of any container with a size that's known at compile-time, such as an array or tuple.

16. “Structured binding declaration.” Accessed May 1, 2021. https://en.cppreference.com/w/cpp/language/structured_binding.

17. Dominik Berner, “Quick and easy unpacking in C++ with structured bindings.” May 24, 2018. Accessed May 4, 2021. <https://dominikberner.ch/structured-bindings/>.

Perf Erase/Remove Idiom

Line 18 uses the **vector**'s **erase** member function to delete the **vector** elements that are no longer valid—this reduces the **vector**'s size to its actual number of elements. Line 20 outputs the updated contents of `v1`.

The `<vector>` header contains common-range algorithms for removing elements from a **vector** and reducing the **vector**'s size—`std::erase`

and `std::erase_if`. Each suffers from a performance problem. As each removes a vector element, it immediately shifts the remaining elements to the beginning of the `vector` and resizes the `vector`. This occurs for every item removed, resulting in many additional passes of the `vector`.

When eliminating elements from a `vector`, it's a good practice to use the **erase-remove idiom**.¹⁸ You remove elements via the `remove` or `remove_if` algorithms, then call the `vector`'s `erase` member function to reduce the container's size. Though the `remove` or `remove_if` algorithms also shift the `vector`'s remaining elements to the beginning of the container, **they do so in a single pass**, rather than multiple passes, making them more efficient than `std::erase` and `std::erase_if`. You can then pass the `iterators` that `remove` or `remove_if` return to the `vector`'s `erase` member function to remove the invalid `vector` elements all at once, rather than once per item erased.

18. “Erase–remove idiom.” Accessed May 5, 2021.
https://en.wikipedia.org/wiki/Erase-remove_idiom.

The **common-range** `std::remove` or `std::remove_if` algorithms each return a single `iterator` pointing to the first invalid element in the `vector`. When using these **common-range algorithms**, you can perform the **erase-remove idiom** in one statement, as in:

[Click here to view code image](#)

```
v1.erase(std::remove(v1.begin(), v1.end(), 10), v1.end());
```

Unfortunately, container member functions like `erase` do not yet support C++20 ranges. So, when working with `std::ranges::remove` and `std::ranges::remove_if`, which return `iterator pairs` representing a `range` of invalid elements, you perform the **erase-remove idiom** in two statements (as in lines 17–18).

[remove_copy Algorithm](#)

20 Concepts Line 28 uses the **C++20 `std::ranges::remove_copy algorithm`** to copy all of `v2`'s elements that do not have the value `10` into the vector `c1`. The first argument must be an `input_range`, which supports `input_iterators` for reading from a

range. Again, a vector supports more powerful **random_access_iterators**, so it meets **remove_copy**'s minimum requirements. We'll discuss the second argument in a moment. Line 30 outputs all of c1's elements.

[Click here to view code image](#)

```
22 std::vector v2{init}; // initialize with co;
23 std::cout << "\n\nv2: ";
24 std::ranges::copy(v2, output);
25
26 // copy from v2 to c1, removing 10s in the ;
27 std::vector<int> c1{};
28 std::ranges::remove_copy(v2, std::back_inse;
29 std::cout << "\nc1 after copying v2 without
30 std::ranges::copy(c1, output);
31
```

< >

```
v2: 10 2 15 4 10 6
c1 after copying v2 without 10s: 2 15 4 6
```

Iterator Adapters: **back_inserter**, **front_inserter** and **inserter**

Concepts The second argument specifies an **output_iterator** indicating where the copied elements will be written. The **remove_copy algorithm** does not check whether the target container has enough room to store all the copied elements. So the **iterator** in the second argument must refer to a container with enough room for all the elements. If you do not want to preallocate memory in advance, you can use an **iterator adaptor** with a **dynamically growable container**, like a **vector**, to insert elements and allow the container to allocate more elements as necessary. In line 28, the expression `std::back_inserter(c1)` uses the **back_inserter iterator adaptor** (`header <iterator>`) to call the container's

push_back function, which inserts an element at the end of the container. If the container needs more space, it grows to accommodate the new element. A **back_inserter** cannot be used with arrays because they are fixed-size containers and do not have a **push_back** function.

Concepts The **remove_copy algorithm** expects as its second argument a **weakly_incrementable iterator**—that is, the **iterator** must support `++` and must allow writing values into the output target, which typically is another container. A **back_inserter** supports these operations.

There are two other **inserters**:

- a **front_inserter** uses the container's **push_front** member function to insert an element at the beginning of its container argument, and
- an **inserter** uses the container's **insert** member function to insert an element into the container specified in its first argument at the location specified by the **iterator** in its second argument.

[remove_if Algorithm](#)

Concepts The C++20 `std::ranges::remove_if algorithm` (line 38) deletes from `v3` all elements for which the **unary predicate function greaterThan9** returns true. The first argument must be a **forward_range** that enables `remove_if` to read the elements in the range. A **unary predicate function** must receive one parameter and return a `bool` value.

[Click here to view code image](#)

```
32     std::vector v3{init}; // initialize with co
33     std::cout << "\n\nv3: ";
34     std::ranges::copy(v3, output);
35
36     // remove elements greater than 9 from a3
37     auto greaterThan9 = [](auto x) {return x >
38         const auto& [first2, last2]{std::ranges::re
```

```
39     v3.erase(first2, last2);
40     std::cout << "\nv3 after removing elements :
41     std::ranges::copy(v3, output);
42
```

```
v3: 10 2 15 4 10 6
v3 after removing elements greater than 9: 2 4 6
```

Line 37 defines a **unary predicate function** as the generic lambda:

```
[] (auto x) {return x > 9;}
```

which returns `true` if its argument is greater than 9; otherwise, it returns `false`. The compiler uses `auto` type inference for both the **lambda**'s parameter and return types:

- The vector contains `ints`, so the compiler infers the parameter's type as `int`.
- The **lambda** returns the result of evaluating a condition, so the compiler infers the return type as `bool`.

Like `remove`, `remove_if` does not modify the number of elements in the container. Instead, it places at the beginning of the container all elements that are not eliminated. It returns an **iterator/sentinel pair** representing the **subrange** of elements that are no longer valid. We use that **iterator/sentinel pair** in line 39 to erase those elements in `v3`. Line 41 outputs the updated contents of `v3`.

remove_copy_if Algorithm

20 Concepts  **Concepts**  The C++20 `std::ranges::remove_copy_if algorithm` (line 49) copies the elements from its `input_range` argument `v4` for which a **unary predicate function** (the **lambda** at line 37) returns `true`. The algorithm's first argument must be an `input_range` so the algorithm can read the `range`'s elements. The second argument must be a **weakly_incrementable iterator** so the element

being copied can be written into the destination container. Here, we again use a **back_inserter** to insert the copied elements into a vector (v4). Line 51 outputs the contents of c2.

[Click here to view code image](#)

```
43     std::vector v4{init}; // initialize with
44     std::cout << "\n\nv4: ";
45     std::ranges::copy(v4, output);
46
47     // copy elements from v4 to c2, removing
48     std::vector<int> c2{}; // initialize to
49     std::ranges::remove_copy_if(v4, std::back_inserter(c2),
50                                std::greater{});
51     std::cout << "\nc2 after copying v4 without elements greater than ";
52     std::ranges::copy(c2, output);
53     std::cout << "\n";
```

< >

```
v4: 10 2 15 4 10 6
c2 after copying v4 without elements greater than
```

< >

14.4.4 **replace**, **replace_if**, **replace_copy** and **replace_copy_if**

Figure 14.5 demonstrates replacing values from a sequence using algorithms **replace**, **replace_if**, **replace_copy** and **replace_copy_if** from C++20's **std::ranges** name-space.

[Click here to view code image](#)

```
1 // fig14_05.cpp
2 // Algorithms replace, replace_if, replace_copy, replace_copy_if
3 #include <algorithm>
```

```
4 #include <array>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7
8 int main() {
9     std::ostream_iterator<int> output{std::cout, " "};
10}
```



Fig. 14.5 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.

replace Algorithm

20 Concepts The **C++20 `std::ranges:: replace`** algorithm (line 16) replaces all elements with the value 10 in the `input_range` `a1`. The first argument must support **indirectly_writeable iterators**, so `replace` can **dereference the iterators** to assign new values to the `range`'s elements. The array `a1` is non-const, so its **iterators** can be used to write into the array.

[Click here to view code image](#)

```
11 std::array a1{10, 2, 15, 4, 10, 6};
12 std::cout << "a1: ";
13 std::ranges::copy(a1, output);
14
15 // replace all 10s in a1 with 100
16 std::ranges::replace(a1, 10, 100);
17 std::cout << "\na1 after replacing 10s with
18 std::ranges::copy(a1, output);
19
```



```
a1: 10 2 15 4 10 6
a1 after replacing 10s with 100s: 100 2 15 4 100
```



[replace_copy Algorithm](#)

20 Concepts C The C++20 `std::ranges:: replace_copy algorithm` (line 26) copies all elements in the `input_range` `a2`, replacing each 10 with 100. The first argument must support `indirectly_copyable iterators`, so `replace_copy` can dereference them to copy to the range's elements. The array `a2` supports `random_access_iterators`, so its `iterators` can be dereferenced to copy the elements. The second argument must be an `output_iterator` so each element can be written into the destination container. The array `c1` is non-const and supports more powerful `random_access_iterators`, so `replace_copy` can write into `c1`. The algorithm copies or replaces every element in the `input_range`, so we allocated `c1` with the same number of elements as `a2`. As shown earlier, we could have used an empty `vector` as the target container and used a `back_inserter` to append elements to the `vector`.

[Click here to view code image](#)

```
20  std::array a2{10, 2, 15, 4, 10, 6};
21  std::array<int, a2.size()> c1{};
22  std::cout << "\n\na2: ";
23  std::ranges::copy(a2, output);
24
25  // copy from a2 to c1, replacing 10s with 1
26  std::ranges::replace_copy(a2, c1.begin(), 1
27  std::cout << "\nc1 after replacing a2's 10s
28  std::ranges::copy(c1, output);
29
```

a2: 10 2 15 4 10 6
c1 after replacing a2

replace_if Algorithm

20 Concepts C The C++20 `std::ranges:: replace_if algorithm` (line 36) replaces each element in the `input_range` `a3` for which a **unary predicate function** (the `lambda greaterThan9` in line 35) returns true. The first argument must support `indirectly_writeable iterators`, so `replace_if` can **dereference the iterators** to assign new values to the `range`'s elements. Here, we replace each value greater than 9 with 100.

[Click here to view code image](#)

```
30 std::array a3{10, 2, 15, 4, 10, 6};
31 std::cout << "\n\na3: ";
32 std::ranges::copy(a3, output);
33
34 // replace values greater than 9 in a3 with
35 constexpr auto greaterThan9 = [](auto x) { r
36 std::ranges::replace_if(a3, greaterThan9, 1
37 std::cout << "\na3 after replacing values g
38 std::ranges::copy(a3, output);
39
```

```
a3: 10 2 15 4 10 6
a3 after replacing values greater than 9 with 100
```

replace_copy_if Algorithm

20 Concepts C Concepts C The C++20 `std::ranges:: replace_copy_if algorithm` (line 46) copies all elements in the `input_range` `a4` for which a **unary predicate function** returns true. Again we replace values greater than 9 with the value 100. The copied or replaced elements are placed in `c2`, starting at position `c2.begin()`, which must be an `output_iterator`. The array `c2` is non-const and

supports more powerful **random_access_iterators**, so **replace_copy_if** can output into `c2`. The algorithm copies or replaces every element in the **input_range**, so we allocated `c2` with the same number of elements as `a4`. The first argument must support **indirectly_copyable iterators**, so **replace_copy_if** can dereference them to copy to the range's elements.

[Click here to view code image](#)

```
40     std::array a4{10, 2, 15, 4, 10, 6};
41     std::array<int, a4.size()> c2{};
42     std::cout << "\n\na4: ";
43     std::ranges::copy(a4, output);
44
45     // copy a4 to c2, replacing elements greater than 9 with 100s
46     std::ranges::replace_copy_if(a4, c2.begin());
47     std::cout << "c2 after replacing a4's elements greater than 9 with 100s: ";
48     std::ranges::copy(c2, output);
49     std::cout << "\n";
50
51 }
```

```
<   >
a4: 10 2 15 4 10 6
c2 after replacing a4
```

14.4.5 Mathematical Algorithms

Figure 14.6 demonstrates several common mathematical algorithms, including **shuffle**, **count**, **count_if**, **min_element**, **max_element**, **minmax_element** and **transform** from C++20's **std::ranges** namespace. There are other mathematical algorithms in the **<numeric>** header, such as the **accumulate** algorithm, which we introduced in Section 8.19.2. The algorithms in that header use **common ranges** and are expected to be updated to **C++20 ranges** in C++23.¹⁹

19. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed May 4, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2214r0.html#algorithms>.

[Click here to view code image](#)

```
1 // fig14_06.cpp
2 // Mathematical algorithms of the standard
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <numeric>
8 #include <random>
9
10 int main() {
11     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9,
12     std::ostream_iterator<int> output{std::cout,
13
14     std::cout << "a1: ";
15     std::ranges::copy(a1, output);
16 }
```

```
a1: 1 2 3 4 5 6 7 8 9 10
```

Fig. 14.6 | Mathematical algorithms of the standard library. (Part 1 of 2.)

shuffle Algorithm

20 Concepts  11 The C++20 `std::ranges::shuffle_algorithm` (line 19) randomly reorders the elements in `a1`, which must be a `random_access_range` that supports `random_access_iterators`. The algorithm’s declaration indicates that the range’s `iterators` must be `permutable`, which permits operations such as swapping and moving elements—`a1` is non-const, so `shuffle` can

perform such operations on the array. The **shuffle** algorithm's second argument is a **C++11 random-number-generator engine**. Line 18

[Click here to view code image](#)

```
default_random_engine randomEngine{random_device{  
    < >
```

11 Sec  creates a **default_random_engine** and initializes it with a C++11 **random_device** object, which seeds the random-number generator—typically with a nondeterministic seed, which cannot be predicted.²⁰ In the expression

20. “Nondeterministic algorithm.” Accessed May 2, 2021. https://en.wikipedia.org/wiki/Nondeterministic_algorithm.

```
random_device{}()
```

the braces initialize the **random_device** object, and the parentheses call its overloaded parentheses operator to get the seed. Line 21 displays the shuffled results.

[Click here to view code image](#)

```
17 // create random-number engine and use it to shuffle  
18 std::default_random_engine randomEngine{std::random_device{}()};  
19 std::ranges::shuffle(a1, randomEngine); // shuffle the array  
20 std::cout << "\na1 shuffled: ";  
21 std::ranges::copy(a1, output);  
22
```

```
< >
```

```
a1 shuffled: 5 4 7 6 3 9 1 8 10 2
```

count Algorithm

20 Concepts  The **C++20 std::ranges:: count algorithm** (line 28) counts the elements with a specific value (in this case, 8) in **a2**,

which must be an **input_range**, so **count** can read elements in the range.

[Click here to view code image](#)

```
23     std::array a2{100, 2, 8, 1, 50, 3, 8, 8, 9,
24     std::cout << "\n\na2: ";
25     std::ranges::copy(a2, output);
26
27     // count number of elements in a2 with value 8
28     auto result1{std::ranges::count(a2, 8)};
29     std::cout << "\nCount of 8s in a2: " << res
30
```

```
a2: 100 2 8 1 50 3 8 8 9 10
Count of 8s in a2: 3
```

[count_if Algorithm](#)

20 Concepts  The **C++20 std::ranges::count_if algorithm** (line 32) counts in its argument **a2** elements for which a **unary predicate** function returns **true**. Once again, we used a **lambda** to define a **unary predicate** that returns **true** for a value greater than 9. The first argument must be an **input_range**, so the algorithm can read elements in the range.

[Click here to view code image](#)

```
31     // count number of elements in a2 that are > 9
32     auto result2{std::ranges::count_if(a2, [](auto a){ return a > 9; });
33     std::cout << "\nCount of a2 elements greater than 9: "
34
```

```
< >
```

Count of a2 elements greater than 9: 3

min_element Algorithm

20 Concepts C Err X The C++20 `std::ranges::min_element algorithm` (line 37) locates the smallest element in its `forward_range` argument `a2`. Such a `range` supports `forward_iterators`, which allow `min_element` to read elements from the `range`. The algorithm returns an `iterator` located at the first occurrence of the `range`'s smallest element or the `range`'s `sentinel` if the `range` is empty. As with many algorithms that compare elements, you can provide a custom `binary predicate function` that specifies how to compare the elements and returns `true` if the first argument is less than the second. Before `dereferencing an iterator` that might represent a `range`'s `sentinel`, you should check that it does not match the `sentinel`.

[Click here to view code image](#)

```
35 // locate minimum element in a2
36 std::cout << "\n\na2 minimum element: "
37     << * (std::ranges::min_element(a2));
38
```

a2 minimum element: 1

max_element Algorithm

20 Concepts C The C++20 `std::ranges::max_element algorithm` (line 41) locates the largest element in its `forward_range` argument `a2`. Such a `range` supports `forward_iterators`, which allow `max_element` to read elements from the `range`. The algorithm returns an `iterator` located at the first occurrence of the `range`'s largest element or the `range`'s `sentinel` if the `range` is empty. You can provide a custom `binary predicate function` that specifies how to compare the elements and returns

true if the first argument is less than the second.

[Click here to view code image](#)

```
39 // locate maximum element in a2
40 std::cout << "\na2 maximum element: "
41     << * (std::ranges::max_element(a2));
42
```

```
a2 maximum element: 100
```

[minmax_element Algorithm](#)

20 Concepts  The C++20 `std::ranges::minmax_element` algorithm (line 44) locates the smallest and largest elements in its `forward_range` argument `a2`. It returns an **iterator pair** aimed at the smallest and largest elements, respectively. If there are duplicate smallest elements, the first **iterator** is located at the **first of the smallest values**. Similarly, if there are duplicate largest elements, the second **iterator** is aimed at the **last of the largest values**. You can provide a custom **binary predicate function** that specifies how to compare the elements and returns true if the first argument is less than the second.

[Click here to view code image](#)

```
43 // locate minimum and maximum elements in a
44 const auto& [min, max]{std::ranges::minmax_
45 std::cout << "\na2 minimum and maximum elem
46     << *min << " and " << *max;
47
```



```
a2 minimum and maximum elements: 1 and 100
```

transform Algorithm

20 Concepts Lines 50–51 use the **C++20 std::ranges::transform algorithm** to transform the elements of its **input_range** `a1` to new values. Each new value is written into the container specified by the algorithm's second argument, which must be an **indirectly_writeable iterator** and can point to the same container as the **input_range** or a different container. The function provided as **transform**'s third argument receives a value and returns a new value and must not modify the elements in the range.²¹

21. “Algorithms library—Mutating sequence operations—Transform.” Accessed May 1, 2021. <https://eel.is/c++draft/alg.transform>.

Click here to view code image

```
48     // calculate cube of each element in a1;
49     std::array<int, a1.size()> cubes{};
50     std::ranges::transform(a1, cubes.begin()
51         [] (auto o x) {return x * x * x; });
52     std::cout << "\n\na1 values cubed: ";
53     std::ranges::copy(cubes, output);
54     std::cout << "\n";
55 }
```

```
a1 values cubed: 125 64 343 216 27 729 1 512 100
```

Concepts A **transform** overload accepts two **input_ranges**, an **indirectly_writeable iterator** and a function that takes two arguments and returns a result. This version of **transform** passes corresponding elements from each **input_range** to its function argument, then outputs the function's result via the **indirectly_writeable iterator**.

14.4.6 Searching and Sorting Algorithms

Figure 14.7 demonstrates some basic searching and sorting algorithms, including `find`, `find_if`, `sort`, `binary_search`, `all_of`, `any_of`, `none_of` and `find_if_not` from C++20's `std::ranges` namespace.

[Click here to view code image](#)

```
1 // fig14_07.cpp
2 // Standard library search and sort algorithm definitions
3 #include <algorithm> // algorithm definitions
4 #include <array> // array class-template definition
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{10, 2, 17, 5, 16, 8, 13, 11, 20, 7};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output); // displays the values
14 }
```



```
values contains: 10 2 17 5 16 8 13 11 20 7
```

Fig. 14.7 | Standard library search and sort algorithms.

find Algorithm

20 Concepts  The C++20 `std::ranges::find` algorithm (line 16) performs an $O(n)$ linear search to find a value (16) in its `input_range` argument (`values`). The algorithm returns an `iterator` that's either positioned at the first element containing the value or indicates the range's `sentinel` (as demonstrated by lines 26–33). We use the returned `iterator` in line 19 to calculate the index position at which the value was

found.

[Click here to view code image](#)

```
15 // locate first occurrence of 16 in values
16 auto loc1{std::ranges::find(values, 16)};
17
18 if (loc1 != values.cend()) { // found 16
19     std::cout << "\n\nFound 16 at index: " <
20 }
21 else { // 16 not found
22     std::cout << "\n\n16 not found";
23 }
24
25 // locate first occurrence of 100 in values
26 auto loc2{std::ranges::find(values, 100)};
27
28 if (loc2 != values.cend()) { // found 100
29     std::cout << "\nFound 100 at index: " <<
30 }
31 else { // 100 not found
32     std::cout << "\n100 not found";
33 }
34
```



```
Found 16 at index:
4 100 not found
```

[find_if Algorithm](#)

20 Concepts The **C++20 `std::ranges::find_if algorithm`** (line 39) performs an **$O(n)$ linear search** to locate the first value in its **input_range** argument (`values`) for which a **unary predicate function** returns true. In this case, we use the **lambda**

`isGreaterThan10` defined in line 36. The algorithm returns an **iterator** that's positioned at the first element containing a value for which the **predicate function** returns `true` or that indicates the sequence's **sentinel**.

[Click here to view code image](#)

```
35 // create variable to store lambda for reuse
36 constexpr auto isGreaterThan10{}[](auto x){return x > 10;}
37
38 // locate first occurrence of value greater than 10
39 auto loc3{std::ranges::find_if(values, isGreaterThan10)};
40
41 if (loc3 != values.cend()) { // found value
42     std::cout << "\n\nFirst value greater than 10: " << *loc3
43     << "\nfound at index: " << (loc3 - values.begin());
44 }
45 else { // value greater than 10 not found
46     std::cout << "\n\nNo values greater than 10 found";
47 }
48
```



sort Algorithm

20 Concepts  The C++20 `std::ranges::sort` algorithm (line 50) performs an **performs an $O(n \log n)$ sort** that arranges the elements in its argument `values` into ascending order. The argument must be a **random_access_range**, which supports **random_access_iterators** and thus can be used with the standard library containers `array`, `vector` and `deque` (and builtin arrays). This algorithm also can receive a **binary predicate function** taking two arguments and returning a `bool` indicating the **sorting order**. The predicate

compares two values from the sequence being sorted. If the return value is true, the two elements are already in sorted order; otherwise, the two elements need to be reordered in the sequence.

[Click here to view code image](#)

```
49 // sort elements of a
50 std::ranges::sort(values);
51 std::cout << "\n\nvalues after sort: ";
52 std::ranges::copy(values, output);
53
```

```
values after sort: 2 5 7 8 10 11 13 16 17 20
```

[binary_search Algorithm](#)

20 Concepts   The C++20 `std::ranges::binary_search algorithm` performs an $O(\log n)$ binary search to determine whether a value (13) is in its `forward_range` argument (values). The range must be sorted in ascending order. The algorithm returns a `bool` indicating whether the value was found in the sequence. Line 63 demonstrates a call to `binary_search` for which the value is not found. This algorithm also can receive a `binary predicate function` with two arguments and returning a `bool`. The function should return `true` if the two elements being compared are in sorted order. If you need to know the search key's location in the container, use the `lower_bound` or `find` algorithms rather than `binary_search`.

[Click here to view code image](#)

```
54 // use binary_search to check whether 13 ex
55 if (std::ranges::binary_search(values, 13))
56     std::cout << "\n\n13 was found in values
57 }
```

```
58     else {
59         std::cout << "\n\n13 was not found in va
60     }
61
62 // use binary_search to check whether 100 e:
63 if (std::ranges::binary_search(values, 100)
64     std::cout << "\n100 was found in values"
65 }
66 else {
67     std::cout << "\n100 was not found in val
68 }
69
```

```
13 was found in values
100 was not found in values
```

[all_of Algorithm](#)

20 Concepts The **C++20 `std::ranges:: all_of`** algorithm (line 71) performs an **$O(n)$ linear search** to determine whether the **unary predicate function** in its second argument (in this case, the **lambda `isGreaterThan10`**) returns true for **all of the elements** in its **input_range** argument (`values`). If so, **all_of** returns true; otherwise, it returns false.

[Click here to view code image](#)

```
70 // determine whether all of values' element
71 if (std::ranges::all_of(values, isGreaterThan10))
72     std::cout << "\n\nAll values elements ar
73 }
74 else {
75     std::cout << "\n\nSome values elements a
76 }
```

 Some values elements are not greater than 10

[any_of Algorithm](#)

20 Concepts The C++20 `std::ranges::any_of` algorithm (line 79) performs an $O(n)$ linear search to determine whether the **unary predicate function** in its second argument (in this case, the **lambda** `isGreaterThan10`) returns true for **at least one of the elements** in its `input_range` argument (values). If so, `any_of` returns true; otherwise, it returns false.

[Click here to view code image](#)

```

78     // determine whether any of values' element
79     if (std::ranges::any_of(values, isGreaterThan10))
80         std::cout << "\n\nSome values elements are not greater than 10"
81     }
82     else {
83         std::cout << "\n\nNo values elements are not greater than 10"
84     }
85

```

 Some values elements are greater than 10

[none_of Algorithm](#)

20 Concepts The C++20 `std::ranges::none_of` algorithm (line 87) performs an $O(n)$ linear search to determine whether the **unary predicate function** in its second argument (in this case, the **lambda** `isGreaterThan10`) returns false for **all of the elements** in

its **input_range** argument (`values`). If so, **none_of** returns true; otherwise, it returns false.

[Click here to view code image](#)

```
86    // determine whether none of values' elemen
87    if (std::ranges::none_of(values, isGreaterT
88        std::cout << "\n\nNo values elements are
89    }
90    else {
91        std::cout << "\n\nSome values elements a
92    }
93
```



Some values elements are greater than 10

[find_if_not Algorithm](#)

20 Concepts  The C++20 `std::ranges:: find_if_not algorithm` (line 95) performs an $O(n)$ linear search to locate the first value in its **input_range** argument (`values`) for which a **unary predicate function** (the `lambda isGreaterThan10`) returns false. The algorithm returns an **iterator** that's either positioned at the first element containing a value for which the **predicate function** returns false or indicates the **range's sentinel**.

[Click here to view code image](#)

```
94    // locate first occurrence of value that
95    auto loc4{std::ranges::find_if_not(value
96
97    if (loc4 != values.cend()) { // found a
98        std::cout << "\n\nFirst value not gre
99        << "\nfound at index: " << (loc4 -
```

```
100      }
101      else { // no values less than or equal
102          std::cout << "\n\nOnly values greater than or equal to 10 are displayed.\n";
103      }
104
105      std::cout << "\n";
106 }
```

```
< >
First value not greater than 10: 2
found at index: 0
```

14.4.7 **swap**, **iter_swap** and **swap_ranges**

Figure 14.8 demonstrates algorithms for swapping elements—**swap** and **iter_swap** from the `std` namespace and algorithm **swap_ranges** from C++20’s `std::ranges` namespace.

[Click here to view code image](#)

```
1 // fig14_08.cpp
2 // Algorithms swap, iter_swap and swap_range
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{1, 2, 3, 4, 5, 6, 7, 8};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14 }
```

```
< >
```

```
values contains: 1 2 3 4 5 6 7 8 9 10
```

Fig. 14.8 | Algorithms swap, iter_swap and swap_ranges.

swap Algorithm

Line 15 uses the `std::swap` algorithm to exchange its two arguments' values—this is not a range or common-range algorithm. The function takes as arguments simply references to two values being exchanged. In this case, we pass references to the array's first and second elements.

[Click here to view code image](#)

```
15     std::swap(values[0], values[1]); // swap elements
16
17     std::cout << "\nvalues after swapping a[0] and a[1]: ";
18     std::ranges::copy(values, output);
19
```

```
values after swapping a[0] and a[1] with swap: 2 1 3 4 5 6 7 8 9 10
```

iter_swap Algorithm

Line 21 uses the `std::iter_swap` algorithm to exchange the two elements specified by its **common-range forward iterator** arguments. The iterators can refer to any two elements of the same type.

[Click here to view code image](#)

```
20 // use iterators to swap elements at locations
21     std::iter_swap(values.begin(), values.begin() + 1);
22     std::cout << "\nvalues after swapping a[0] and a[1]: ";
23     std::ranges::copy(values, output);
24
```

```
values after swapping a[0] and a[1] with iter_swap
```

swap_ranges Algorithm

20 Concepts C Concepts C Line 31 uses the C++20 `std::ranges:: swap_ranges algorithm` to exchange the elements of its two `input_range` arguments. If the `ranges` are not the same length, the algorithm swaps the shorter sequence with the corresponding elements in the longer sequence. The `ranges` also must support `indirectly_swappable iterators`, so the algorithm can `dereference the iterators` to swap to the corresponding elements in each `range`.

[Click here to view code image](#)

```
25 // swap values and values2
26 std::array values2{10, 9, 8, 7, 6, 5, 4, 3,
27 std::cout << "\n\nBefore swap_ranges\nvalues"
28 std::ranges::copy(values, output);
29 std::cout << "\nvalues2 contains: ";
30 std::ranges::copy(values2, output);
31 std::ranges::swap_ranges(values, values2);
32 std::cout << "\n\nAfter swap_ranges\nvalues"
33 std::ranges::copy(values, output);
34 std::cout << "\nvalues2 contains: ";
35 std::ranges::copy(values2, output);
36
```

```
Before swap_ranges
values contains: 1 2 3 4 5 6 7 8 9 10
values2 contains: 10 9 8 7 6 5 4 3 2 1
```

```
After swap_ranges
```

```
values contains: 10 9 8 7 6 5 4 3 2 1  
values2 contains: 1 2 3 4 5 6 7 8 9 10
```

20 Concepts C Lines 38–39 use the C++20

`std::ranges::swap_ranges` overload that enables you to specify the portions of two `input_ranges` to swap. Here, we swap `values`'s first five elements with its last five elements, specifying the two ranges to swap as **iterator pairs**:

- `values.begin()` and `values.begin() + 5` indicate the first five elements, and
- `values.begin() + 5` and `values.end()` indicate the last five elements.

Specifying **iterator pairs** also works with the **common ranges version** in the `std` name-space. In fact, **most common ranges algorithms require iterator pairs**. In this example, the two **ranges** are in the same container, but **the ranges can be from different containers. The ranges must not overlap**.

[Click here to view code image](#)

```
37      // swap first five elements of values and last five of values2
38      std::ranges::swap_ranges(values.begin(),
39                                values2.begin(), values2.begin() + 5)
40
41      std::cout << "\n\nAfter swap_ranges for 5 elements\n";
42      std::ranges::copy(values, output);
43      std::cout << "values contains: ";
44      std::ranges::copy(values2, output);
45      std::cout << "\nvalues2 contains: ";
46      std::cout << "\n";
47 }
```



```
After swap_ranges for 5 elements
values contains: 1 2 3 4 5 5 4 3 2 1
```

```
values2 contains: 10 9 8 7 6 6 7 8 9 10
```

14.4.8 `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`

20 Figure 14.9²² demonstrates algorithms `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n` from C++20's `std::ranges` namespace.

22. As of May 2021, `std::ranges::unique` did not compile on the most recent clang++ compiler.

[Click here to view code image](#)

```
1 // fig14_09.cpp
2 // Algorithms copy_backward, merge, unique,
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <vector>
8
9 int main() {
10     std::array a1{1, 3, 5, 7, 9};
11     std::array a2{2, 4, 5, 7, 9};
12     std::ostream_iterator<int> output{std::cout, " "};
13
14     std::cout << "array a1 contains: ";
15     std::ranges::copy(a1, output); // displays 1 3 5 7 9
16     std::cout << "\narray a2 contains: ";
17     std::ranges::copy(a2, output); // displays 2 4 5 7 9
18 }
```

```
< >
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9
```

Fig. 14.9 | Algorithms `copy_backward`, `merge`, `unique` and `reverse`.

`copy_backward` Algorithm

20 Concepts The `C++20 std::ranges::copy_backward algorithm` (line 21) copies its first argument's `bidirectional_range` (`a1`) into the destination specified by its second argument, which indicates the `end of the target container` (`results.end()`). The algorithm copies the original elements in reverse order, placing each element into the target container, **starting from the element before `results.end()` and working toward the beginning of the container**. The `iterator` specified by the second argument must be `indirectly_copyable`, so `copy_backward` can **dereference the iterator** to copy the value it references. The algorithm returns an `iterator pair` in which the first is positioned at `a1.end()`, and the second is positioned at the last element copied into the target container—that is, the beginning of `results` because the copy is performed backwards. Though the elements are copied in reverse order, they're placed in `results` in the same order as `a1`. One difference between `copy` and `copy_backward` is that

- the `iterator` returned from `copy` is positioned *after* the last element copied, and
- the one returned from `copy_backward` is positioned *at* the last element copied (i.e., the first element in the sequence).

[Click here to view code image](#)

```
19 // place elements of a1 into results in rev
20 std::array<int, a1.size()> results{};
21 std::ranges::copy_backward(a1, results.end());
22 std::cout << "\n\nAfter copy_backward, resu
23 std::ranges::copy(results, output);
24
```



```
After copy_backward, results contains: 1 3 5 7 9
```



move and move_backward Algorithm

20 You can use **move semantics** with ranges of elements. The **C++20 std::ranges algorithms move and move_backward** (from header <algorithm>) work like the **copy** and **copy_backward algorithms**, but move the elements in the specified ranges rather than copying them.

merge Algorithm

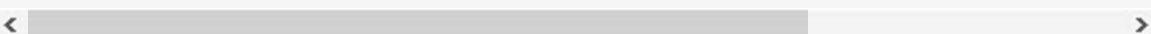
20 The **C++20 std::ranges:: merge algorithm** (line 27) combines two **input_ranges** that are each **sorted in ascending order** to the target container specified by the third argument's **weakly_incrementable output iterator**. After this operation, **results2** contains both **ranges**' values in sorted order. A second version of **merge** takes **iterator/sentinel pairs** representing the two **ranges**. Both versions allow you to provide a **binary predicate function** that specifies the sorting order by comparing its two arguments and returning **true** if the first is less than the second.

[Click here to view code image](#)

```
25 // merge elements of a1 and a2 into results
26 std::array<int, a1.size() + a2.size()> resu
27 std::ranges::merge(a1, a2, results2.begin())
28
29 std::cout << "\n\nAfter merge of a1 and a2,
30 std::ranges::copy(results2, output);
31
```



```
After merge of a1 and a2, results2 contains: 1 2
```



unique Algorithm

20 Concepts C The C++20 `std::ranges::unique` algorithm (line 34) determines the unique values in the sorted range of values specified by its `forward_range` argument. After `unique` is applied to a `sorted range` with duplicate values, a single copy of each value remains in the `range`. The algorithm returns an **iterator pair**. The first is positioned *after* the last unique value, and the second is positioned *at* the end of the original `range`. The values of all elements in the container after the last unique value are undefined. They should not be used, so this is another case in which you can erase the unused elements (line 35). You also may provide a **binary predicate function** specifying how to compare two elements for equality.

[Click here to view code image](#)

```
32 // eliminate duplicate values from v
33 std::vector v(results2.begin(), results2.end());
34 const auto& [first, last] {std::ranges::unique(
35     v.erase(first, last); // remove elements that
36
37     std::cout << "\n\nAfter unique v contains:
38     std::ranges::copy(v, output);
39
```

After unique, v contains: 1 2 3 4 5 7 9

reverse Algorithm

20 Concepts C The C++20 `std::ranges::reverse` algorithm (line 41) reverses the elements in its argument—a `bidirectional_range` that supports `bidirectional_iterators`.

[Click here to view code image](#)

```
40     std::cout << "\n\nAfter reverse, a1 contains:\n";
41     std::ranges::reverse(a1); // reverse elements
42     std::ranges::copy(a1, output);
43
```

< >

After reverse, a1 contains: 9 7 5 3 1

copy_if Algorithm

11 20 Concepts C++11 added the copy algorithms **copy_if** and **copy_n**, and C++20 added **ranges** versions of each. The **C++20 std::ranges:: copy_if algorithm** (lines 47–48) receives as arguments an **input_range**, a **weakly_incrementable output iterator** and a **unary predicate function**. The algorithm calls the **unary predicate function** for each element in the **input_range** and copies only those elements for which the function returns `true`. The **output iterator** specifies where to output elements—in this case, we use a **back_inserter** to add elements to a `vector`. The algorithm returns an **iterator pair**—the first is positioned *at* the end of the **input_range**, and the second is positioned *after* the last element copied into the output container.

[Click here to view code image](#)

```
44 // copy odd elements of a2 into v2
45 std::vector<int> v2{ };
46 std::cout << "\n\nAfter copy_if, v2 contains:\n";
47 std::ranges::copy_if(a2, std::back_inserter(v2),
48                     [] (auto x){return x % 2 == 0;});
49 std::ranges::copy(v2, output);
50
```

< >

```
After copy_if, v2 contains: 2 4
```

copy_n Algorithm

20 Concepts The C++20 `std::ranges::copy_n` algorithm (line 54) copies from the location specified by the `input_iterator` in its first argument (`a2.begin()`) the number of elements specified by its second argument (3). The elements are output to the location specified by the `weakly_incrementable output iterator` in the third argument—in this case, we use a `back_inserter` to add elements to a vector.

[Click here to view code image](#)

```
51     // copy three elements of a2 into v3
52     std::vector<int> v3{ };
53     std::cout << "\n\nAfter copy_n, v3 conta
54     std::ranges::copy_n(a2.begin(), 3, std::b
55     std::ranges::copy(v3, output);
56     std::cout << "\n";
57 }
```

```
< >
```

```
After copy_n, v3 contains: 2 4 5
```

14.4.9 `inplace_merge`, `unique_copy` and `reverse_copy`

Figure 14.10²³ demonstrates algorithms `inplace_merge`, `unique_copy` and `reverse_copy` from C++20's `std::ranges` namespace.

²³ As of May 2021, `std::ranges::unique_copy` did not compile on the most recent Visual C++ compiler.

[Click here to view code image](#)

```
1 // fig14_10.cpp
2 // Algorithms inplace_merge, reverse_copy and
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <vector>
8
9 int main() {
10     std::array a1{1, 3, 5, 7, 9, 1, 3, 5, 7,
11     std::ostream_iterator<int> output(std::cout);
12
13     std::cout << "array a1 contains: ";
14     std::ranges::copy(a1, output);
15 }
```

< >

```
array a1 contains: 1 3 5 7 9 1 3 5 7 9
```

Fig. 14.10 | Algorithms `inplace_merge`, `reverse_copy` and `unique_copy`. (Part 1 of 2.)

`inplace_merge` Algorithm

20 Concepts C The C++20 `std::ranges:: inplace_merge` algorithm (line 18) merges two sorted sequences of elements in its `bidirectional_range` argument. This example processes the `range` `a1` (the first argument), **merging** the elements from `a1.begin()` up to, but not including, `a1.begin() + 5` (the second argument) with the elements starting from `a1.begin() + 5` (the second argument) up to, but not including the end of the `range`. You also can pass a **binary predicate function** that compares elements in the two `subranges` and returns `true` if the first is less than the second.

[Click here to view code image](#)

```
16 // merge first half of a1 with second half
17 // a1 contains sorted set of elements after
18 std::ranges::inplace_merge(a1, a1.begin() +
19 std::cout << "\nAfter inplace_merge, a1 con-
20 std::ranges::copy(a1, output);
21
```

< >

After inplace_merge, a1 contains: 1 1 3 3 5 5 7

< >

unique_copy Algorithm

20 Concepts The C++20 `std::ranges:: unique_copy algorithm` (line 24) copies the unique elements in its first argument's sorted `input_range`. The `weakly_incrementable` `output iterator` supplied as the second argument specifies where to place the copied elements—in this case, the `back_inserter` adds new elements in the vector `results1`, growing it as necessary. You also can pass a `binary predicate function` for comparing elements for equality.

[Click here to view code image](#)

```
22 // copy only unique elements of a1 into res-
23 std::vector<int> results1{};
24 std::ranges::unique_copy(a1, std::back_inse-
25 std::cout << "\nAfter unique_copy, results1
26 std::ranges::copy(results1, output);
27
```

< >

After unique_copy results1 contains: 1 3 5 7 9

Fig. 14.10 | Algorithms `inplace_merge`, `reverse_copy` and

`unique_copy`. (Part 2 of 2.)

[reverse_copy Algorithm](#)

20 The C++20 `std::ranges:: reverse_copy algorithm` (line 30) makes a reversed copy of its first argument's `bidirectional_range`. The `weakly_incrementable output iterator` supplied as the second argument specifies where to place the copied elements—in this case, the `back_inserter` adds new elements in the vector `results2`, growing it as necessary.

[Click here to view code image](#)

```
28     // copy elements of a1 into results2 in
29     std::vector<int> results2{};
30     std::ranges::reverse_copy(a1, std::back_
31     std::cout << "\nAfter reverse_copy, resu
32     std::ranges::copy(results2, output);
33     std::cout << "\n";
34 }
```

```
<          >
After reverse_copy, results2 contains: 9 9 7 7 5
```

14.4.10 Set Operations

20 Figure 14.11 demonstrates the C++20 `std::ranges` namespace's set-manipulation `algorithms` `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union`. In addition to the capabilities described in this section, you can customize these algorithms' element comparisons by passing a `binary predicate function` that compares two elements to determine whether the first is less than the second.

[Click here to view code image](#)

```
1 // fig14_11.cpp
2 // Algorithms includes, set_difference, set_
3 // set_symmetric_difference and set_union.
4 #include <array>
5 #include <algorithm>
6 #include <fmt/format.h> // C++20: This will
7 #include <iostream>
8 #include <iterator>
9 #include <vector>
10
11 int main() {
12     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9,
13     std::array a2{4, 5, 6, 7, 8};
14     std::array a3{4, 5, 6, 11, 15};
15     std::ostream_iterator<int> output{std::cout, " "};
16
17     std::cout << "a1 contains: ";
18     std::ranges::copy(a1, output); // displays 1 2 3 4 5 6 7 8 9
19     std::cout << "\na2 contains: ";
20     std::ranges::copy(a2, output); // displays 4 5 6 7 8
21     std::cout << "\na3 contains: ";
22     std::ranges::copy(a3, output); // displays 4 5 6 11 15
23 }
```

< >

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
```

Fig. 14.11 | Algorithms includes, set_difference, set_intersection, set_symmetric_difference and set_union.

includes Algorithm

20 Concepts C The C++20 `std::ranges:: includes`

algorithm (lines 26 and 31) compares two sorted **input_ranges** to determine whether every element of the second is in the first. The **ranges must be sorted** using the same **comparison function**. The algorithm returns true if all the second **range**'s elements are in the first; otherwise, it returns false. In line 26, a2's elements are all in a1, so **includes** returns true. In line 31, a3's elements are not all in a1, so the **includes** returns false.

[Click here to view code image](#)

```
24 // determine whether a2 is completely conta
25 std::cout << fmt::format("\n\na1 {} a2",
26                         std::ranges::includes(a1, a2),
27                         "includes" : "does not i:
28
29 // determine whether a3 is completely conta
30 std::cout << fmt::format("\n\na1 {} a3",
31                         std::ranges::includes(a1, a3),
32                         "includes" : "does not i:
33
```



```
a1 includes a2
a1 does not include a3
```

[set_difference Algorithm](#)

20 Concepts  **Concepts**  The C++20 **std::ranges::set_difference algorithm** (line 36) finds the elements from the first sorted **input_range** that are not in the second sorted **input_range**. The **ranges** must be sorted using the same **comparison function**. The elements that differ are copied to the location specified by the **weakly_incrementable output iterator** supplied as the third argument —in this case, a **back_inserter** adds them to the vector difference.

[Click here to view code image](#)

```
34 // determine elements of a1 not in a2
35 std::vector<int> difference{};
36 std::ranges::set_difference(a1, a2, std::back_inserter(difference));
37 std::cout << "\n\nset_difference of a1 and a2 is: ";
38 std::ranges::copy(difference, output);
39
```



```
set_difference of a1 and a2 is: 1 2 3 9 10
```

[set_intersection Algorithm](#)

20 Concepts C Concepts C The C++20 `std::ranges::set_intersection algorithm` (lines 42–43) determines the elements from the first sorted `input_range` that are in the second sorted `input_range`. The `ranges` must be sorted using the same **comparison function**. The elements common to both are copied to the location specified by the **weakly_incrementable output iterator** supplied as the third argument—in this case, a `back_inserter` adds them to the vector intersection.

[Click here to view code image](#)

```
40 // determine elements in both a1 and a2
41 std::vector<int> intersection{};
42 std::ranges::set_intersection(a1, a2,
43     std::back_inserter(intersection));
44 std::cout << "\n\nset_intersection of a1 and a2 is: ";
45 std::ranges::copy(intersection, output);
46
```



```
set_intersection of a1 and a2 is: 4 5 6 7 8
```

`set_symmetric_difference` Algorithm

20 Concepts C++20 `std::ranges::`

The `set_symmetric_difference` algorithm (lines 50–51) determines the elements in the first sorted `input_range` that are not in the second sorted `input_range` and the elements in the second `range` that are not in the first. The `ranges` must be sorted using the same **comparison function**. Each `input_range`'s elements that are different are copied to the location specified by the **weakly_incrementable output iterator** supplied as the third argument—in this case, a `back_inserter` adds them to `symmetricDifference`.

[Click here to view code image](#)

```
47 // determine elements of a1 that are not in
48 // elements of a3 that are not in a1
49 std::vector<int> symmetricDifference{};
50 std::ranges::set_symmetric_difference(a1, a
51     std::back_inserter(symmetricDifference))
52 std::cout << "\n\nset_symmetric_difference"
53 std::ranges::copy(symmetricDifference, outp
54
```

< >

```
set_symmetric_difference of a1 and a3 is: 1 2 3
```

< >

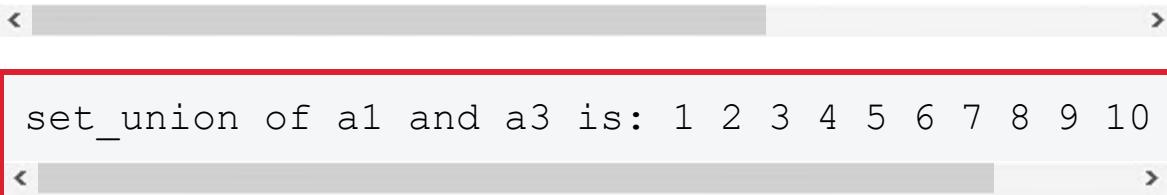
`set_union` Algorithm

20 Concepts C++20 `std::ranges::` `set_union` algorithm (line 57) creates a set of all the elements that are in either or both of its two sorted `input_ranges`, which must be sorted using the same **comparison function**. The elements are copied to the location specified by the **weakly_incrementable output iterator** supplied as the third

argument—in this case, a **back_inserter** adds them to `unionSet`. Elements that appear in both sets are copied only from the first set.

[Click here to view code image](#)

```
55     // determine elements that are in either
56     std::vector<int> unionSet{};
57     std::ranges::set_union(a1, a3, std::back_
58     std::cout << "\n\nset_union of a1 and a3"
59     std::ranges::copy(unionSet, output);
60     std::cout << "\n";
61 }
```



```
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10
```

14.4.11 `lower_bound`, `upper_bound` and `equal_range`

Figure 14.12²⁴ demonstrates the algorithms `lower_bound`, `upper_bound` and `equal_range` from C++20's `std::ranges` namespace. In addition to the capabilities described in this section, you can customize these algorithms' element comparisons by passing a **binary predicate function** that compares two elements to determine whether the first is less than the second.

24. As of May 2021, `std::ranges::equal_range` did not compile on the most recent Clang compiler.

[Click here to view code image](#)

```
1 // fig14_12.cpp
2 // Algorithms lower_bound, upper_bound and
3 // equal_range for a sorted sequence of val-
4 #include <algorithm>
5 #include <array>
6 #include <iostream>
```

```
7 #include <iterator>
8
9 int main() {
10     std::array values{2, 2, 4, 4, 4, 6, 6, 6
11     std::ostream_iterator<int> output{std::cout,
12
13     std::cout << "values contains: ";
14     std::ranges::copy(values, output);
15 }
```

```
Array a contains: 2 2 4 4 4 6 6 6 6 8
```

Fig. 14.12 | Algorithms `lower_bound`, `upper_bound` and `equal_range` for a sorted sequence of values.

lower_bound Algorithm

20 Concepts  The **C++20 `std::ranges::lower_bound algorithm`** (line 17) finds in a **sorted forward_range** the first location at which the second argument could be inserted such that the **range** would still be **sorted in ascending order**. The algorithm returns an **iterator** pointing to that location.

[Click here to view code image](#)

```
16 // determine lower-bound insertion point for
17 auto lower{std::ranges::lower_bound(values,
18 std::cout << "\n\nLower bound of 6 is index
19 << (lower - values.begin());
```

```
Lower bound of 6 is element 5 of array a
```

upper_bound Algorithm

20 Concepts C The C++20 `std::ranges:: upper_bound algorithm` (line 22) finds in a **sorted forward_range** the last location at which the second argument could be inserted such that the range would still be **sorted in ascending order**. The algorithm returns an **iterator** pointing to that location.

[Click here to view code image](#)

```
21 // determine upper-bound insertion point fo
22 auto upper{std::ranges::upper_bound(values,
23 std::cout << "\nUpper bound of 6 is index:
24 << (upper - values.begin());
```

```
25
```


Upper bound of 6 is element 9 of array a

equal_range Algorithm

20 The C++20 `std::ranges:: equal_range algorithm` (line 27) returns a subrange (name-space `std::ranges`) containing two **iterators** representing the results of performing both a **lower_bound** and an **upper_bound** operation. Line 27 uses **structured bindings** to unpack these **iterators** into `first` and `last`, which we use to calculate the corresponding index positions in `values`.

[Click here to view code image](#)

```
26 // use equal_range to determine the lower a
27 const auto& [first, last]{std::ranges::equa
28 std::cout << "\nUsing equal_range:\n Lower "
29 << (first - values.begin());
30 std::cout << "\n Upper bound of 6 is index:
```

```
31     << (last - values.begin());  
32
```

Using `equal_range`:

Lower bound of 6 is element 5 of array a
Upper bound of 6 is element 9 of array a

Locating Insertion Points in Sorted Sequences

Algorithms `lower_bound`, `upper_bound` and `equal_range` are often used to locate a new value's insertion point in a sorted sequence. Line 36 uses `lower_bound` to locate the first position at which 3 can be inserted in order in `values`. Line 43 uses `upper_bound` to locate the last point at which 7 can be inserted in order in `values`.

[Click here to view code image](#)

```
33 // determine lower-bound insertion point  
34 std::cout << "\n\nUse lower_bound to loc.  
35     << "at which 3 can be inserted in ord.  
36 lower = std::ranges::lower_bound(values,  
37 std::cout << "\n Lower bound of 3 is ind.  
38     << (lower - values.begin());  
39  
40 // determine upper-bound insertion point  
41 std::cout << "\n\nUse upper_bound to loc.  
42     << "at which 7 can be inserted in ord.  
43 upper = std::ranges::upper_bound(values,  
44 std::cout << "\n Upper bound of 7 is ind.  
45     << (upper - values.begin());  
46 }
```

Use `lower_bound` to locate the first point at which 5 can be inserted in order

Lower bound of 5 is element 5 of array a

Use `upper_bound` to locate the last point at which 7 can be inserted in order

Upper bound of 7 is element 9 of array a

14.4.12 `min`, `max` and `minmax`

Figure 14.13 demonstrates algorithms `min`, `max` and `minmax` from the `std` namespace and the `minmax` overload from C++20's `std::ranges namespace`. Unlike the algorithms we presented in Section 14.4.5, which operated on `ranges`, the `std` namespace's `min`, `max` and `minmax` algorithms operate on two values passed as arguments. The `std::ranges::minmax` algorithm returns the minimum and maximum values in a `range`.

Algorithms `min` and `max` with Two Parameters

The `min` and `max` algorithms (demonstrated in lines 8–12) each receive two arguments and return the minimum or maximum value. Note that capital is *less than lowercase*.

[Click here to view code image](#)

```
1 // fig14_13.cpp
2 // Algorithms min, max, minmax and minmax_e
3 #include <array>
4 #include <algorithm>
5 #include <iostream>
6
7 int main() {
8     std::cout << "Minimum of 12 and 7 is: "
9         << "\nMaximum of 12 and 7 is: " << std::
10        << "\nMinimum of 'G' and 'Z' is: '" <<
11        << "\nMaximum of 'G' and 'Z' is: '" <<
12        << "\nMinimum of 'z' and 'Z' is: '" <<
13}
```

```
< >  
Minimum of 12 and 7 is: 7  
Maximum of 12 and 7 is: 12  
Minimum of  
Maximum of  
Minimum of
```

Fig. 14.13 | Algorithms `min`, `max`, `minmax` and `minmax_element`.

C++11 `minmax` Algorithm with Two Arguments

11 C++11 added the two-argument **minmax algorithm** (line 15), which returns a **pair of values** containing the smaller and larger item, respectively. Here we used **structured bindings** to unpack the values into `smaller` and `larger`. A second version of `minmax` takes as a third argument a **binary predicate function** for performing a **custom comparison** that determines whether the first argument is less than the second.

[Click here to view code image](#)

```
14 // determine which argument is the min and  
15 auto [smaller, larger]{std::minmax(12, 7)};  
16 std::cout << "\n\nMinimum of 12 and 7 is: "  
17           << "\nMaximum of 12 and 7 is: " << larg  
18
```

```
< >  
The minimum of 12 and 7 is: 7  
The maximum of 12 and 7 is: 12
```

`minmax` Algorithm for C++20 Ranges

20 Concepts The C++20 `std::ranges::minmax` algorithm (line 25) returns a pair of values containing the minimum and

maximum items in its `input_range` or `initializer_list` argument. Here, we used **structured bindings** to unpack these values into smallest and largest, respectively. You also can pass as a second argument a **binary predicate function** for comparing elements to determine whether the first is less than the second.

[Click here to view code image](#)

```
19     std::array items{3, 100, 52, 77, 22, 31,
20     std::ostream_iterator<int> output{std::cout,
21
22     std::cout << "\n\nitems: ";
23     std::ranges::copy(items, output);
24
25     const auto& [smallest, largest] = std::minmax(items);
26     std::cout << "\nMinimum value in items: "
27             << "\nMaximum value in items is: " <<
28 }
```

```
<          >
items: 3 100 52 77 22 31 1 98 13 40
Minimum value in items: 1
Maximum value in items is: 100
```

14.4.13 Algorithms `gcd`, `lcm`, `iota`, `reduce` and `partial_sum` from Header `<numeric>`

We've introduced the `accumulate` algorithm from header `<numeric>`. Figure 14.14 demonstrates several additional `<numeric>` algorithms —`gcd`, `lcm`, `iota`, `reduce` and `partial_sum`. This header's algorithms do not yet have C++20 `std::ranges` versions.

gcd Algorithm

The **gcd algorithm** (lines 14 and 15) receives two integer arguments and returns their greatest common divisor.

[Click here to view code image](#)

```
1 // fig14_14.cpp
2 // Demonstrating algorithms gcd, lcm, iota,
3 #include <array>
4 #include <algorithm>
5 #include <functional>
6 #include <iostream>
7 #include <iterator>
8 #include <numeric>
9
10 int main() {
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     // calculate the greatest common divisor
14     std::cout << "std::gcd(75, 20): " << std::gcd(75, 20)
15     << "\nstd::gcd(17, 13): " << std::gcd(17, 13)
16 }
```

```
< >
std::gcd(75, 20): 5
std::gcd(17, 13): 1
```

Fig. 14.14 | Demonstrating algorithms gcd, lcm, iota, reduce and partial_sum.

lcm Algorithm

The **lcm algorithm** (lines 18 and 19) receives two integer arguments and returns their least common multiple.

[Click here to view code image](#)

```
17 // calculate the least common multiple of two integers
18 std::cout << "\n\nstd::lcm(3, 5): " << std::lcm(3, 5)
19 << "\nstd::lcm(12, 9): " << std::lcm(12, 9)
```

20

```
< >  
std::lcm(3, 5): 15  
std::lcm(12, 9): 36
```

iota Algorithm

The **iota algorithm** (line 23) fills a **common range** with a sequence of values starting with the value in the third argument. The first two arguments must be **forward iterators** indicating the beginning and end of a **common range** in which the element type (in this case, `int`) must support incrementing values with `++`.

[Click here to view code image](#)

```
21 // fill an array with integers using the std::iota algorithm  
22 std::array<int, 5> ints{};  
23 std::iota(ints.begin(), ints.end(), 1);  
24 std::cout << "\nints: " ;  
25 std::ranges::copy(ints, output);  
26
```

```
< >  
ints: 1 2 3 4 5
```

reduce Algorithm

The **reduce algorithm** (lines 29 and 31) enables you to reduce a **common range**'s elements to a single value. The first and second arguments must be **input iterators**. The call in line 29 implicitly adds the **common range**'s elements. The call in line 31 enables you to provide a custom initializer value (1) and a **binary function** that specifies how to perform the **reduction**. In this case, we used `std::multiplies{}`—a predefined **binary function object**²⁵ from the header `<functional>` that multiplies its two arguments and returns the result. The `{}` create a temporary `std::multiplies`

object and call its constructor. Every **function object** has an **overloaded operator() function**. Inside the **reduce** algorithm, it calls function **operator()** on the **function object** to produce a result. Any binary function that takes two values of the same type and returns a result of that type can be passed as the fourth argument. As you'll see in [Section 14.5](#), header `<functional>` defines **binary function objects** for addition, subtraction, multiplication, division and modulus, among others.

25. We say more about the predefined function objects in [Section 14.5](#).

[Click here to view code image](#)

```
27 // reduce elements of a container to a sing
28 std::cout << "\n\nsum of ints: "
29     << std::reduce(ints.begin(), ints.end())
30     << "\nproduct of ints: "
31     << std::reduce(ints.begin(), ints.end(),
32
```



```
sum of ints: 15
product of ints: 120
```

[reduce vs. accumulate](#)

The **reduce** algorithm is similar to the **accumulate** algorithm **but does not guarantee the order in which the elements are processed**—in [Chapter 17](#), you'll see that **this difference in operation is why the reduce algorithm can be parallelized for better performance, but the accumulate algorithm cannot**.²⁶

26. Sy Brand, “`std::accumulate` vs. `std::reduce`,” May 15, 2018. Accessed May 6, 2021.
<https://blog.tartanllama.xyz/accumulate-vs-reduce/>.

[partial_sum Algorithm](#)

The **partial_sum algorithm** (lines 37 and 39) calculates a partial sum of its **common range**'s elements from the start of the range through the current element. By default, this version of **partial_sum** uses the **std::plus**

function object, which adds its two arguments and returns their sum. For `ints`, which line 23 filled with the values 1, 2, 3, 4 and 5, the call in line 37 outputs the following sums:

- 1 (this is simply the value of `ints`' first element),
- 3 (the sum $1 + 2$),
- 6 (the sum $1 + 2 + 3$),
- 10 (the sum $1 + 2 + 3 + 4$), and
- 15 (the sum $1 + 2 + 3 + 4 + 5$).

[Click here to view code image](#)

```
33     // calculate the partial sums of ints' elements
34     std::cout << "\n\nints: ";
35     std::ranges::copy(ints, output);
36     std::cout << "\n\npartial_sum of ints using std::plus by default: ";
37     std::partial_sum(ints.begin(), ints.end());
38     std::cout << "\npartial_sum of ints using std::multiplies: ";
39     std::partial_sum(ints.begin(), ints.end(),
40                      std::multiplies{});
41 }
```

```
<          >
ints: 1 2 3 4 5

partial_sum of ints using std::plus by default:

partial_sum of ints using std::multiplies: 1 2 6
```

The second call (line 39) uses the `partial_sum` overload that receives a **binary function** specifying how to perform partial calculations. In this case, we used the predefined **binary function object** `std::multiplies{}`, resulting in the products of the values from the beginning of the container to the current element:

- 1 (this is simply the value of `ints`' first element),
- 2 (the product $1 * 2$)
- 6 (the product $1 * 2 * 3$)
- 24 (the product $1 * 2 * 3 * 4$)
- 120 (the product $1 * 2 * 3 * 4 * 5$)

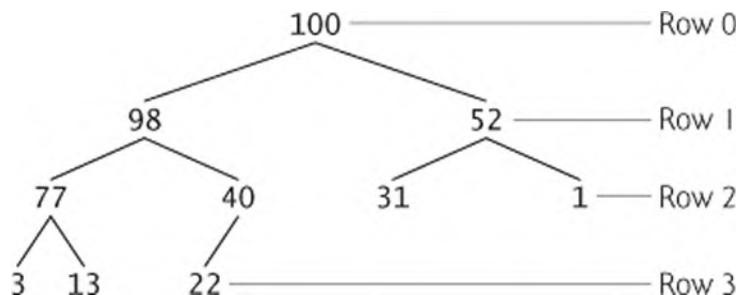
Because `ints` contains the values 1–5, the result of this `partial_sum` call is actually calculating $1!$, $2!$, $3!$, $4!$ and $5!$ (that is, the factorials of 1–5).

14.4.14 Heapsort and Priority Queues

Section 13.10.3 introduced the `priority_queue` container adaptor. Elements added to a `priority_queue` are stored in a manner that enables removing them in **priority order**. The highest-priority element—that is, the one with the largest value—is always removed first. This can be done efficiently by arranging the elements in a data structure called a **heap**—not to be confused with the heap C++ maintains for dynamic memory allocation. A common uses of priority queues are in operating system process scheduling, triaging hospital emergency room patients and vaccination scheduling during the Covid-19 pandemic.

Heap Data Structure

A **heap** is commonly implemented as a **binary tree data structure**. A **max heap** stores its largest value in the root node, and **any given child node's value is less than or equal to its parent node's value**. Heaps may contain duplicate values. A **heap** also can be a **min heap** in which the smallest value is in the root node, and any given child node's value is greater than or equal to its parent node's value. The following diagram shows a binary tree representing a **max heap**:



A **heap** is typically stored in an array-like data structure, such as an **array**,

vector or **deque**, each of which uses **random-access iterators**. The following diagram shows the preceding diagram's max heap represented as an array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
100	98	52	77	40	31	1	3	13	22					

Row 0 Row 1 Row 2 Row 3

You can confirm that this array represents a max heap. For any given array index n , you can find the parent node's array index by calculating

$$(n - 1) / 2$$

using integer arithmetic. Then you can confirm that the child node's value is less than or equal to the parent node's value. A **max heap**'s largest value is always at the top of the binary tree, which corresponds to the array's first element.

Heap-Related Algorithms

20 Figure 14.15 demonstrates four C++20 `std::ranges` namespace algorithms related to **heaps**. First, we show `make_heap` and `heap_sort`, which implement the two steps in the **heapsort algorithm**, which has a worst-case runtime of $O(n \log n)$:²⁷

27. "Heapsort." Accessed May 8, 2021. <https://en.wikipedia.org/wiki/Heapsort>.

- arranging the elements of a container into a **heap**, then
- removing the elements from the **heap** to produce a sorted sequence.

Then, we show `push_heap` and `pop_heap`, which the **priority_queue container adaptor**²⁸ uses “under the hood” to maintain its elements in a **heap** as elements are added and removed.

28. You can see the open source Microsoft C++ standard library implementation of `priority_queue` using `push_heap` and `pop_heap` at <https://github.com/microsoft/STL/blob/main/stl/inc/queue>. Accessed May 10, 2021.

[Click here to view code image](#)

1 // fig14_15.cpp

```
2 // Algorithms push_heap, pop_heap, make_heap
3 #include <iostream>
4 #include <algorithm>
5 #include <array>
6 #include <vector>
7 #include <iterator>
8
9 int main() {
10     std::ostream_iterator<int> output(std::cout,
11
```



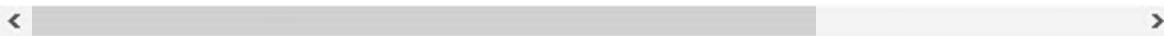
Fig. 14.15 | Algorithms push_heap, pop_heap, make_heap and sort_heap.

Initializing and Displaying heapArray

Line 12 creates and initializes the array heapArray with 10 different unsorted integers. Line 14 displays heapArray before we convert its contents to a **heap** and **sort** the elements.

[Click here to view code image](#)

```
12 std::array heapArray{3, 100, 52, 77, 22, 31
13 std::cout << "heapArray before make_heap:\n"
14 std::ranges::copy(heapArray, output);
15
```



```
heapArray before make_heap:
3 100 52 77 22 31 1 98 13 40
```

make_heap Algorithm

The **C++20 std::ranges:: make_heap algorithm** (line 16) arranges the elements of its **random_access_range** argument into a **heap**, which can then be used with **sort_heap** to produce a sorted

sequence. Line 18 displays heapArray with its elements arranged in a **heap**. A **random_access_range** supports **random_access_iterators**, so this algorithm works with **arrays**, **vectors** and **deques**.

[Click here to view code image](#)

```
16 std::ranges::make_heap(heapArray); // creates a heap
17 std::cout << "\nheapArray after make_heap:\\";
18 std::ranges::copy(heapArray, output);
19
```

< >

```
heapArray after make_heap:
100 98 52 77 40 31 1 3 13 22
```

[sort_heap Algorithm](#)

The **C++20 std::ranges:: sort_heap algorithm** (line 20) sorts the elements of its **random_access_range** argument. The **range** must already be a **heap**. Line 22 displays the sorted heapArray.

[Click here to view code image](#)

```
20 std::ranges::sort_heap(heapArray); // sorts the heap
21 std::cout << "\nheapArray after sort_heap:\\";
22 std::ranges::copy(heapArray, output);
23
```

< >

```
heapArray after sort_heap:
1 3 13 22 31 40 52 77 98 100
```

[Using push_heap and pop_heap to Maintain a Heap](#)

Next, we'll demonstrate the algorithms a **priority_queue** uses "under the hood" to **insert a new item in a heap** and **remove an item from a heap**. Both operations are $O(\log n)$.²⁹ Lines 25–33 define the lambda push, which adds one `int` value to a heap that's stored in a `vector`. To do so, push performs the following tasks:

29. "Binary heap." Accessed May 10, 2021.
https://en.wikipedia.org/wiki/Binary_heap.

- Line 28 appends one `int` to the `vector` argument `heap`.

20 • Line 29 calls the **C++20 `std::ranges::push_heap algorithm`**, which takes the last element of its **random_access_range** argument (`heap`) and inserts it into the **heap data structure**. Each time `push_heap` is called, it assumes that the **range**'s last element is the one being added to the **heap** and that the **range**'s other elements are already arranged as a **heap**. If the element appended in line 28 is the **range**'s only element, the **range** is already a **heap**. Otherwise, `push_heap` rear-ranges the elements into a **heap**.

- Line 31 displays the updated **heap data structure** after each value is added.

[Click here to view code image](#)

```
24 // lambda to add an int to a heap
25 auto push{
26     [&] (std::vector<int>& heap, int value) {
27         std::cout << "\n\npushing " << value
28         heap.push_back(value); // add value to
29         std::ranges::push_heap(heap); // inser
30         std::cout << "\nheap: ";
31         std::ranges::copy(heap, output);
32     }
33 };
34
```

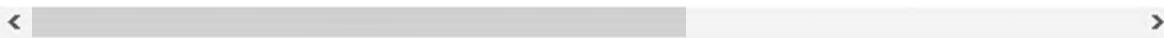


Lines 36–44 define the lambda `pop`, which removes the largest value from the heap data structure. To do so, `pop` performs the following tasks:

- 20 • Line 38 uses the `C++20 std::ranges:: pop_heap algorithm` to remove the largest value from the heap. The algorithm assumes that its `random_access_range` argument's elements represent a **heap data structure**. First, it swaps the largest **heap** element (located at `heap.begin()`) with the last heap element (the one before `heap.end()`). Then, it ensures that the elements from the `range`'s beginning up to, but not including, the `range`'s last element still form a **heap**. **The pop_heap algorithm does not modify the number of elements in the range.**
- Line 39 displays the value of the `vector`'s last element—the value that was just removed from the **heap** but still remains in the `vector`.
 - Line 40 removes the `vector`'s last element, leaving only the `vector` elements that still represent a **heap data structure**.
 - Line 42 shows the current **heap data structure** contents

[Click here to view code image](#)

```
35 // lambda to remove an item from the heap
36 auto pop{
37     [&] (std::vector<int>& heap) {
38         std::ranges::pop_heap(heap); // removes largest
39         std::cout << "\n\npopping highest pri-
40         heap.pop_back(); // remove vector's l-
41         std::cout << "\nheap: ";
42         std::ranges::copy(heap, output);
43     }
44 };
45
```



Demonstrating a Heap Data Structure

Line 46 defines an empty `vector<int>` in which we'll maintain the **heap**

data structure. Lines 49–51 call the **push lambda** to add the values 3, 100 and 52 to the heap. As we add each value, note that the largest value is always stored in the **vector**'s first element and that the elements are not stored in sorted order.

[Click here to view code image](#)

```
46     std::vector<int> heapVector{};  
47  
48     // place five integers into heapVector, mainly  
49     for (auto value : {3, 52, 100}) {  
50         push(heapVector, value);  
51     }  
52
```

< >

```
pushing 3 onto heap  
heap: 3
```

```
pushing 52 onto heap  
heap: 52 3
```

```
pushing 100 onto heap  
heap: 100 3 52
```

Next, line 53 calls the **pop lambda** removes the highest-priority item (100) from the **heap**. Note that the largest remaining value (52) is now in the **vector**'s first element. Line 54 adds the value 22 to the **heap**.

[Click here to view code image](#)

```
53     pop(heapVector); // remove max item  
54     push(heapVector, 22); // add new item to heap  
55
```

< >

```
popping highest priority item: 100  
heap: 52 3
```

```
pushing 22 onto heap  
heap: 52 3 22
```

Next, line 56 removes the highest-priority item (52) from the **heap**. Again, the largest remaining value (22) is now in the **vector**'s first element. Line 57 adds the value 77 to the **heap**. This is now the largest value, so it becomes the **vector**'s first element.

[Click here to view code image](#)

```
56     pop(heapVector); // remove max item  
57     push(heapVector, 77); // add new item to he  
58
```

```
< >  
popping highest priority item: 52  
heap: 22 3
```

```
pushing 77 onto heap  
heap: 77 3 22
```

Finally, lines 59–61 remove the three remaining items in the **heap**. Note that after line 59 executes, the largest remaining element (22) becomes the **vector**'s first element.

[Click here to view code image](#)

```
59     pop(heapVector); // remove max item  
60     pop(heapVector); // remove max item  
61     pop(heapVector); // remove max item  
62     std::cout << "\n";
```

```
 popping highest priority item: 77
heap: 22 3
```

```
 popping highest priority item: 22
heap: 3
```

```
 popping highest priority item: 3
heap:
```

14.5 Function Objects (Functors)

As we've shown, many standard library algorithms allow you to pass a **lambda** or a **function pointer** into the algorithm to help it perform its task. Any algorithm that can receive a **lambda** or **function pointer** can also receive an object of a class that overloads the function-call operator (parentheses) with a function named **operator()**, provided that the overloaded operator meets the algorithm's requirements for number of parameters and return type. For instance, the **transform** algorithm requires a **unary function** that has a single parameter and returns the transformed value.

An object of a class that overloads function **operator()** is known as a **function object** or a **functor** and can be used syntactically and semantically like a **lambda** or a **function pointer**. The **operator() function** is invoked by using the object's name followed by parentheses containing the arguments. Most algorithms can use lambdas, function pointers and function objects interchangeably.

Benefits of Function Objects

Function objects provide several benefits over functions and pointers to functions:

- A **function object** is an object of a class. As such, it can have non-static data members to maintain state for a specific **function object**, or static data members to maintain state shared by all function objects

of that class type. Also, the class type of a **function object** can be used as a default type argument for template type parameters.³⁰

30. “Function Objects in the C++ Standard Library.” March 15, 2019. Accessed May 6, 2021. <https://docs.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl>.

Perf  • Perhaps most importantly, the compiler can inline **function objects** for performance. A **function object's operator()** function typically is defined in its class's body, making it implicitly inline. When defined outside its class's body, function **operator()** can be declared `inline` explicitly. Compilers also convert **lambdas** to **function objects**, so these, too, can be inlined. On the other hand, compilers typically do not inline functions invoked via function pointers —such pointers could be aimed at any function with the appropriate parameters and return type, so a compiler does not know which function to inline.³¹

31. Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. p.201–202: Pearson Education, 2001.

Predefined Function Objects of the Standard Template Library

20 Many **predefined function objects** can be found in the header `<functional>`. Each is implemented as a class template. The following table lists some of the commonly used standard library **function objects**. Each of the **relational function objects** has a corresponding one of the same name in the **C++20 std::ranges namespace**.

Function object	Type	Function object	Type
<code>divides<T></code>	arithmetic	<code>logical_or<T></code>	logical
<code>equal_to<T></code>	relational	<code>minus<T></code>	arithmetic
<code>greater<T></code>	relational	<code>modulus<T></code>	arithmetic
<code>greater_equal<T></code>	relational	<code>negate<T></code>	arithmetic
<code>less<T></code>	relational	<code>not_equal_to<T></code>	relational
<code>less_equal<T></code>	relational	<code>plus<T></code>	arithmetic
<code>logical_and<T></code>	logical	<code>multiplies<T></code>	arithmetic
<code>logical_not<T></code>	logical		

Most **function objects** in the preceding table are **binary function objects** that receive two arguments and return a result—`logical_not` and `negate` are **unary function objects** that receive one argument and return a result. For example, consider the following function object for comparing two `int` values:

```
std::less<int> smaller{ };
```

We can use the object's name (`smaller`) to call its **operator() function**, as follows:

```
smaller(10, 7)
```

Here, `smaller` would return `false` because 10 is not less than 7—an algorithm like `sort` would use this information to reorder these values. A **unary function object's operator() function** receives only one argument.

You can see the complete list of **function objects** in the `<functional> header` at

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/utility/function>



and in the “Function Objects” section of the C++ standard.³² We used the

function object `less<T>` in Section 13.9’s presentations of `set` and `multiset` to specify the sorting order for elements in a container. The `std::ranges algorithms` that compare elements for ordering use as their **default predicate function argument** the **function object** `less<T>`. Recall that many of the overloaded standard library algorithms that perform comparisons can receive a **binary function** that determines whether its first argument is less than its second—exactly the purpose of the `less<T> function object`.

32. “General utilities library—Function objects.” Accessed May 2, 2021. <https://eel.is/c++draft/function.objects>.

Using the `accumulate` Algorithm

Figure 14.16 uses the `std::accumulate numeric algorithm (header <numeric>)` to calculate the sum of the squares of an array’s elements. The `<numeric>` algorithms do not have C++20 `std::ranges overloads`, so they use **common ranges**—`std::ranges` overloads of these algorithms are proposed for C++23.³³ The `accumulate` algorithm has two overloads. The three-argument version adds the **common range**’s elements by default. The four-argument version receives as its last argument a **binary function** that customizes how to perform the calculation. That argument can be supplied as:

33. Christopher Di Bella, “A Concept Design for the Numeric Algorithms,” August 2, 2019. Accessed May 5, 2021. <http://wg21.link/p1813r0>.

- a **function pointer** to a **binary function** that takes two parameters of the **common range**’s type and returns a result of that type,
- a **binary function object** in which the `operator()` function takes two parameters of the **common range**’s type and returns a result of that type, or
- a **lambda** that takes two parameters of the **common range**’s type and returns a result of that type.

This example calls `accumulate` with a **function pointer**, then a **function object** and then a **lambda**.

[Click here to view code image](#)

```
1 // fig14_16.cpp
2 // Demonstrating function objects.
3 #include <array>
4 #include <algorithm>
5 #include <functional>
6 #include <iostream>
7 #include <iterator>
8 #include <numeric>
9
```

Fig. 14.16 | Demonstrating function objects.

Function sumSquares

Lines 12–14 define a **function sumSquares** that takes two arguments of the same type and returns a value of that type—the requirements for the **binary function** that **accumulate** can receive as an argument. The sumSquares function returns the sum of its first argument total and the square of its second argument value.

[Click here to view code image](#)

```
10 // binary function returns the sum of its f
11 // and the square of its second argument va
12 int sumSquares(int total, int value) {
13     return total + value * value;
14 }
15
```



Class SumSquaresClass

Lines 19–25 define the **class SumSquaresClass**.³⁴ Its **overloaded operator() function** has two **int** parameters and returns an **int**. This meets requirements for the **binary function** that **accumulate** can call when processing a **common range** of **int** values. The **operator() function** returns the sum of its first argument total and the square of its

second argument value.

34. This class handles only `int` values, but could be implemented as a class template that handles many types—we'll define custom class templates in [Chapter 15](#).

[Click here to view code image](#)

```
16 // class SumSquaresClass defines overloaded
17 // that returns the sum of its first argume:
18 // and the square of its second argument va
19 class SumSquaresClass {
20 public:
21     // add square of value to total and retu
22     int operator()(int total, int value) {
23         return total + value * value;
24     }
25 };
26
```



Calling Algorithm `accumulate`

We call `accumulate` three times:

- Lines 36–37 call `accumulate` with a **pointer to function** `sumSquares` as its last argument.
- Lines 44–45 call `accumulate` with a **function object of class `SumSquaresClass`** as the last argument. The expression `SumSquaresClass{}` in line 45 creates a **temporary `SumSquaresClass` object** and calls its constructor. That **function object** is then passed to `accumulate`, which calls the **`SumSquaresClass` object's `operator()` function**.
- Lines 50–51 call `accumulate` with an equivalent **lambda**. The **lambda** performs the same tasks as the **function `sumSquares`** and the overloaded **`operator()` function in `SumSquaresClass`**.

[Click here to view code image](#)

```
27 int main() {
28     std::array integers{1, 2, 3, 4};
29     std::ostream_iterator<int> output{std::cout, " "};
30
31     std::cout << "array integers contains: ";
32     std::ranges::copy(integers, output);
33
34     // calculate sum of squares of elements
35     // using binary function sumSquares
36     int result{std::accumulate(integers.cbegin(),
37                               0, sumSquares)};
38
39     std::cout << "\n\nSum of squares\n"
40             << "via binary function sumSquares: ";
41
42     // calculate sum of squares of elements
43     // using binary function object
44     result = std::accumulate(integers.cbegin(),
45                             0, SumSquaresClass{});
46
47     std::cout << "\nvia a SumSquaresClass fu-
48
49     // calculate sum of squares array
50     result = std::accumulate(integers.cbegin(),
51                             0, [](auto total, auto value){return
52
53     std::cout << "\nvia a lambda: " << result;
54 }
```

< >

array integers contains: 1 2 3 4

Sum of squares

via binary function sumSquares: 30

via a SumSquaresClass function object: 30

via a lambda: 30

In each of the three calls to **accumulate**, it uses its function argument as follows:

- On the first call to its function argument, **accumulate** passes its third argument's value (0 in this example) and the value of `integers`' first element (1 in this example). This calculates and returns the result of $0 + 1 * 1$, which is 1.
- On the second call to its function argument, **accumulate** passes the preceding result (1) and the value of `integers`' next element (2). This calculates and returns the result of $1 + 2 * 2$, which is 5.
- On the third call to its function argument, **accumulate** passes the preceding result (5) and the value of `integers`' next element (3). This calculates and returns the result of $5 + 3 * 3$, which is 14.
- On the last call to its function argument, **accumulate** passes the preceding result (14) and the value of `integers`' next element (4). This calculates and returns the result of $14 + 16$, which is 30.

At this point **accumulate** reaches the end of the **common range** specified by its first two arguments, so it returns the result (30) of the last call to its function argument.

14.6 Projections

20 When working on objects that each contain multiple data items, each C++20 **std::ranges algorithm** can use a **projection** to select a narrower part of each object to process. Consider `Employee` objects that each have a first name, a last name and a salary. Rather than sorting `Employees` based on all three data members, **you can sort them using only their salaries**. [Figure 14.17](#) sorts an array of `Employee` objects by salary—first in **ascending order**, then in **descending order**. Lines 11–22 define class `Employee`. Lines 25–29 provide an **overloaded operator<<** function for `Employees` so we can output them conveniently.

[Click here to view code image](#)

```
1 // fig14_17.cpp
2 // Demonstrating projections with C++20 range algorithms
3 #include <array>
4 #include <algorithm>
5 #include <fmt/format.h>
6 #include <iostream>
7 #include <iterator>
8 #include <string>
9 #include <string_view>
10
11 class Employee {
12 public:
13     Employee(std::string_view first, std::string_view last,
14             int salary) : m_first{first}, m_last{last}, m_salary{salary} {}
15     std::string getFirst() const { return m_first; }
16     std::string getLast() const { return m_last; }
17     int getSalary() const { return m_salary; }
18 private:
19     std::string m_first;
20     std::string m_last;
21     int m_salary;
22 };
23
24 // operator<< for an Employee
25 std::ostream& operator<<(std::ostream& out,
26                             const Employee& e) {
27     out << fmt::format("{:10}{:10}{:10}",
28                         e.getLast(), e.getFirst(), e.getSalary());
29     return out;
30 }
```



Fig. 14.17 | Demonstrating projections with C++20 range algorithms.

Defining and Displaying an `array<Employee>`

Lines 32–36 define an `array<Employee>` and line 41 displays its

contents, so we can confirm later that the Employees are sorted properly.

[Click here to view code image](#)

```
31 int main() {
32     std::array employees{
33         Employee{ "Jason", "Red", 5000 },
34         Employee{ "Ashley", "Green", 7600 },
35         Employee{ "Matthew", "Indigo", 3587 }
36     };
37
38     std::ostream_iterator<Employee> output{ s
39
40     std::cout << "Employees:\n";
41     std::ranges::copy(employees, output);
42 }
```



```
Employees:
Red      Jason      5000
Green    Ashley     7600
Indigo   Matthew   3587
```

Using a Projection to Sort the `array<Employee>` in Ascending Order

20 Lines 45–46 calls the `std::ranges::sort` algorithm, passing three arguments:

- The first argument (`employees`) is the **range** to **sort**.
- The second argument (`{}`) is the **binary predicate function** that the **sort algorithm** uses to compare elements when determining their **sort order**. The notation `{}` indicates that `sort` should use the **default binary predicate function** specified in `sort`'s definition—the `std::ranges::less` function object. This causes `sort` to arrange the elements in **ascending order**. The **less function object** compares its two arguments and returns `true` if the first is less than the second. If

`less` returns `false`, the two salaries are not in ascending order, so `sort` reorders the corresponding `Employee` objects.

- The last argument specifies the **projection**. This **unary function** receives an element from the `range` and returns a portion of that element. Here, we implemented the **unary function** as a **lambda** that returns its `Employee` argument's salary. The **projection** is applied *before* `sort` compares the elements, so rather than comparing entire `Employee` objects to determine their sort order, `sort` compares only the Employees' salaries.

[Click here to view code image](#)

```
43     // sort Employees by salary; {} indicates
44     // use its default comparison function
45     std::ranges::sort(employees, {},
46                         [] (const auto& e) {return e.getSalary();
47                         });
48     std::cout << "\nEmployees sorted in ascending order by salary:" <<
49     std::ranges::copy(employees, output);
```



```
Employees sorted in ascending order by salary:
Indigo      Matthew    3587
Red         Jason     5000
Green       Ashley    7600
```

Shorthand Notation for a Projection

We can replace the unary function that we implemented as a **lambda** in line 46 with the shorthand notation

```
&Employee::getSalary
```

This creates a pointer to the `Employee` class's `getSalary` member function, shortening lines 45–46 to:

[Click here to view code image](#)

```
std::ranges::sort(employees, {}, &Employee::getSa
```



To be used as a **projection**, the member function must be `public` and must have no parameters, because the `std::ranges` algorithms cannot receive additional arguments to pass to the member function specified in the **projection**.³⁵

35. “Under the hood,” `sort` uses the `std::invoke` function (header `<functional>`) to call `getSalary` on each `Employee` object.

A Projection Can Be a Pointer to a public Data Member

If a class has a `public` data member, you can pass a pointer to it as the **projection argument**. For example, if our `Employee` class had a `public` data member named `salary`, we could specify `sort`’s **projection argument** as

```
&Employee::salary
```

Using a Projection to Sort the array<Employee> in Descending Order

In algorithms like `sort` that have function arguments, you can combine custom functions and **projections**. For example, lines 51–52 specify both a **binary predicate function object** and a **projection to sort** `Employees` in **descending order** by `salary`. Again, we pass three arguments:

- The first argument (`employees`) is the **range** to sort.
- The second argument creates a `std::ranges::greater` **function object**. This causes `sort` to arrange the elements in **descending order**. The **greater function object** compares its two arguments and returns `true` if the first is greater than the second. If `greater` returns `false`, the two salaries are not in descending order, so `sort` reorders the corresponding `Employee` objects.
- The last argument is the **projection**. So, the `std::ranges::greater` **function object** will compare the `int` salaries of `Employees` to **determine the sort order**.

[Click here to view code image](#)

```
50     // sort Employees by salary in descending order
51     std::ranges::sort(employees, std::ranges::
52                         &Employee::getSalary);
53     std::cout << "\nEmployees sorted in descending order:";
54     std::ranges::copy(employees, output);
55 }
```

```
< >
```

```
Employees sorted in descending order by salary:
Green      Ashley      7600
Red        Jason       5000
Indigo     Matthew     3587
```

```
< >
```

20 14.7 C++20 Views and Functional-Style Programming

In [Section 6.14.3](#), we showed **views** performing operations on **ranges**. We showed that **views** are **composable**, so you can **chain them together** to process a **range**'s elements through a **pipeline of operations**. A **view** does not have its own copy of a **range**'s elements—it simply moves the elements through a **pipeline of operations**. **Views** are one of C++20's **key functional-style programming** capabilities.

Perf  The algorithms we've presented in this chapter so far are **greedy**—when you call an algorithm, it immediately performs its specified task. You saw in [Section 6.14.3](#) that views are **lazy**—they do not produce results until you iterate over them in a loop or pass them to an algorithm that iterates over them. As we discussed in [Section 6.14.3](#), lazy evaluation produces values on demand, which can reduce your program's memory consumption and improve performance when all the values are not needed at once.

14.7.1 Range Adaptors

[Section 6.14.3](#) also demonstrated functional-style **filter** and **map** operations using:

- `std::views::filter` to keep only those `view` elements for which a `predicate`

`function` returns `true`, and

- `std::views::transform` to `map` each `view` element to a new value, possibly of a different type.

20 These are each **range adaptors** (defined in the **header** `<ranges>`). Each takes a `range` as an argument and returns a `std::ranges::viewable_range`, which is commonly referred to as a `view`. The following table lists all the C++20 **range adaptors**. These can be used with the **functional-style programming** techniques we introduced in [Section 6.14](#). Range adaptors are defined in the `<ranges>` header as part of the namespace `std::ranges::views`. The `<ranges>` header also defines the alias `std::views` for `std::ranges::views`.

Range adaptor	Description
filter	Creates a view representing only the range elements for which a predicate returns true.
transform	Creates a view that maps elements to new values.
common	Used to convert a view into a std::ranges::common_range , which enables a range to be used with common-range algorithms that require begin/end iterator pairs of the same type.
all	Creates a view representing all of a range 's elements.
counted	Creates a view of a specified number of elements from either the beginning of a range or from a specific iterator position.
reverse	Creates a view for processing a bidirectional view in reverse order.
drop	Creates a view that ignores the specified number of elements at the beginning of another view .
drop_while	Creates a view that ignores the elements at the beginning of another view as long as a predicate returns true.
take	Creates a view containing the specified number of elements from the beginning of another view .
take_while	Creates a view containing the elements from the beginning of another view until a predicate becomes false.
join	Creates a view that combines the elements of multiple ranges.
split	Splits a view based on a delimiter. The new view contains a separate view for each subrange.
keys	Creates a view of the keys in key-value pairs, such as those in a map . The keys are the first elements in the pairs.
values	Creates a view of the values in key-value pairs, such as those in a map . The values are the second elements in the pairs.
elements	For a view containing tuples , pairs or arrays , creates a view consisting of elements from a specified index in each object.

14.7.2 Working with Range Adaptors and Views

Figure 14.18 demonstrates several `std::views` from the preceding table and introduces the infinite version of the `std::views::iota` range factory. Line 13 defines a **lambda** that returns true if its argument is an even integer. We'll use this **lambda** in our **pipelines**.

[Click here to view code image](#)

```
1 // fig14_18.cpp
2 // Working with C++20 std::views.
3 #include <algorithm>
4 #include <iostream>
5 #include <iterator>
6 #include <map>
7 #include <ranges>
8 #include <string>
9 #include <vector>
10
11 int main() {
12     std::ostream_iterator<int> output{std::cout};
13     auto isEven{ [](int x) {return x % 2 == 0; }
14 }
```



Fig. 14.18 | Working with C++20 `std::views`.

Creating an Infinite Range With `std::views::iota`

In Fig. 6.13, we introduced `std::views::iota` which is known as a **range factory**—it's a view that **lazily creates a sequence of consecutive integers** when you iterate over it. The version of `iota` in Fig. 6.13 requires two arguments—the starting integer value and the value that's one past the end of the sequence that `iota` should produce. Line 16 uses `iota`'s **infinite range** version, which receives only the starting integer (0) in the sequence and increments it by one until you tell it to stop—you'll see how momentarily. The **pipeline** in line 16 creates a **view** that **filters** the

integers produced by `iota`, keeping only the integers for which the `lambda` `isEven` returns true. At this point, no integers have been produced. Recall that **views are lazy**—they do not execute until you iterate through them with a loop or a standard library algorithm. Views are objects that you can store in variables so you can reuse their processing steps and even add more processing steps later. We store this `view` in `evens`. We'll use `evens` to build several enhanced **pipelines**.

[Click here to view code image](#)

```
15      // infinite view of even integers starting at 0
16      auto evens{std::views::iota(0) | std::views::filter(isEven)};
17
```



take Range Adaptor

Though an **infinite range** is logically infinite,³⁶ to process one in a loop or pass one to a standard library algorithm, **you must first limit the number of elements the pipeline will produce**; otherwise, your program will contain an **infinite loop**. There are several **range adaptors** that can limit the number of elements to process—they work with **infinite ranges** and **fixed-size ranges**. Line 19 uses the **take range adaptor** to take only the first five values from the **evens pipeline** that we defined in line 16. We pass the resulting **view** to `std::ranges::copy` (line 19), which iterates through the **pipeline**, causing it to execute its steps:

³⁶. Jeff Garland, “Using C++20 Ranges Effectively,” Jun 18, 2019. Accessed May 2, 2021.
<https://www.youtube.com/watch?v=VmWS-9idT3s>.

- `iota` produces an integer,
- `filter` checks if it's even and, if so,
- `take` passes that value to `copy`.

If the value `iota` produces is not even, `filter` discards that value, and `iota` produces the next value in the sequence. This process repeats until `take` has passed the specified number of elements to `copy`, which displays the results using the `ostream_iterator` output. **You can take any**

number of items from an infinite range or up to the maximum number of items in a fixed-size range. For demonstration purposes, we'll process just a few items in each of the subsequent pipelines we discuss.

[Click here to view code image](#)

```
18     std::cout << "First five even ints: ";
19     std::ranges::copy(evens | std::views::take(
20
```

```
<          >
First five even ints: 0 2 4 6 8
```

take_while Range Adaptor

Lines 22–23 create an enhanced **pipeline** that uses the **take_while range adaptor** to limit the **evens infinite range**. This **range adaptor** returns a **view** that takes elements from the earlier steps in the **pipeline** until **take_while**'s **unary predicate** returns false. In this case, we take even integers while those values are less than 12. The first value greater than or equal to 12 terminates the **pipeline**. We store the **view** returned by **take_while** in the variable **lessThan12** for use in subsequent statements. Line 24 passes **lessThan12** to **std::ranges::copy**, which iterates through the **view**—executing its **pipeline** steps—and displays the results.

[Click here to view code image](#)

```
21     std::cout << "\nEven ints less than 12: ";
22     auto lessThan12{
23         evens | std::views::take_while([](int x)
24             std::ranges::copy(lessThan12, output);
25
```

```
<          >
```

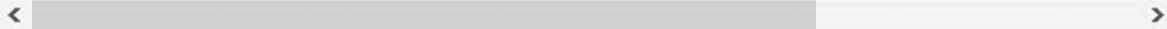
```
Even ints less than 12: 0 2 4 6 8 10
```

reverse Range Adaptor

Line 27 uses the **reverse range adaptor** to reverse the integers from the **view** `lessThan12` from lines 22–23. We pass the **view** returned by **reverse** to `std::ranges::copy`, which iterates through the **view**—executing its **pipeline** steps—and displays the results.

[Click here to view code image](#)

```
26     std::cout << "\nEven ints less than 12 reve
27     std::ranges::copy(lessThan12 | std::views:::
28
```



```
Even ints less than 12 reversed: 10 8 6 4 2 0
```

transform Range Adaptor

We introduced the **transform range adaptor** in Fig. 6.13. The **pipeline** in lines 31–33 creates a **view** that gets the integers produced by `lessThan12`, reverses them then uses **transform** to square their values. We pass the resulting **view** to `std::ranges::copy`, which iterates through the **view**—executing its **pipeline** steps—and displays the results.

[Click here to view code image](#)

```
29     std::cout << "\nSquares of even ints less t:
30     std::ranges::copy(
31         lessThan12
32             | std::views::reverse
33             | std::views::transform([](int x) { re
34         output);
35
```



```
Squares of even ints less than 12 reversed: 100
```

drop Range Adaptor

The **pipeline** in line 38 begins with the **infinite sequence** of even integers produced by **evens**, uses the **drop range adaptor** to skip the first 1000 even integers, then uses the **take range adaptor** to take the next five even integers in the sequence. We pass the resulting **view** to **std::ranges::copy**, which iterates through the **view**—executing its **pipeline** steps—and displays the results.

[Click here to view code image](#)

```
36     std::cout << "\nSkip 1000 even ints, then t.  
37     std::ranges::copy(  
38         evens | std::views::drop(1000) | std::vi  
39         output);  
40
```

```
Skip 1000 even ints, then take five: 2000 2002 2
```

drop_while Range Adaptor

You also can skip elements while a **unary predicate** remains true. The **pipeline** in lines 43– 45 begins with the **infinite sequence** of even integers produced by **evens**, uses the **drop_while range adaptor** to skip even integers at the beginning of the **infinite sequence** that are less than or equal to 1000, then uses the **take range adaptor** to take the next five even integers in the sequence. We pass the resulting **view** to **std::ranges::copy**, which iterates through the **view**—executing its **pipeline** steps—and displays the results.

[Click here to view code image](#)

```
41     std::cout << "\nFirst five even ints greater than 1000: ";
42     std::ranges::copy(
43         evens
44         | std::views::drop_while([](int x) { return x <= 1000; })
45         | std::views::take(5),
46         output);
47 
```

```
< >
First five even ints greater than 1000: 1002 1004 1006 1008 1010
```

```
< >
```

Creating and Displaying a map of Roman Numerals and Their Decimal Values

So far, we've focused on processing **ranges** of integers for simplicity, but you can process **ranges** of more complex types and process various **containers** as well. Next, we'll process the **key-value pairs** in a **map** containing string objects as **keys** and **ints** as **values**. The **keys** are roman numerals and the **values** are their corresponding decimal values. The using declaration in line 49 enables us to use **string object literals** as we build each **key-value pair** in line 52. For example, in the value "I"s, the s following the string literal designates that the literal is a string object. Lines 53–54 create a **lambda** that we use in line 56 with **std::ranges::foreach** to display each **key-value pair** in the **map**.

[Click here to view code image](#)

```
48 // allow std::string object literals
49 using namespace std::literals::string_literals;
50
51 std::map<std::string, int> romanNumerals{
52     {"I"s, 1}, {"II"s, 2}, {"III"s, 3}, {"IV"s, 4},
53     auto displayPair{[] (const auto& p) {
```

```
54     std::cout << p.first << " = " << p.second;
55     std::cout << "\nromanNumerals:\n";
56     std::ranges::for_each(romanNumerals, display);
57 }
```

```
romanNumerals:
I = 1
II = 2
III = 3
IV = 4
V = 5
```

keys and values Range Adaptors

When working with **maps** in pipelines, each **key-value pair** is treated as a **pair** object containing a **key** and a **value**. You can use the **range adaptors** **keys** (line 60) and **values** (line 63) to get **views** of only the **keys** and **values**, respectively:

- **keys** creates a **view** that selects only the *first* item in each **pair**, and
- **values** creates a **view** that selects only the *second* item in each **pair**.

We pass each **pipeline** to **std::ranges::copy**, which iterates through the **pipeline** and displays the results

[Click here to view code image](#)

```
58     std::ostream_iterator<std::string> stringOutput;
59     std::cout << "\nKeys in romanNumerals: ";
60     std::ranges::copy(romanNumerals | std::views::keys,
61                      stringOutput);
62     std::cout << "\nValues in romanNumerals: ";
63     std::ranges::copy(romanNumerals | std::views::values,
64                      stringOutput);
```

```
Keys in romanNumerals: I II III IV V  
Values in romanNumerals: 1 2 3 4 5
```

elements Range Adaptor

Interestingly, the **keys** and **values** range adaptors also work with ranges in which each element is a **tuple** or an **array**. Even if they contain more than two elements each, **keys** always selects the *first* item and **values** always selects the *second*. If the **tuple** or **array** elements in the **range** have more than two elements, the **elements range adaptor** can select items by index, as we demonstrate with `romanNumerals`' **key-value pairs**. Line 67 selects only *element 0* from each **pair** object in the **range**, and line 68 selects only *element 1*. In both cases, we pass the pipeline to **std::ranges::copy**, which iterates through the pipelines and displays the results

[Click here to view code image](#)

```
65     std::cout << "\nKeys in romanNumerals via  
66     std::ranges::copy(  
67         romanNumerals | std::views::elements<  
68     );  
69     std::cout << "\nvalues in romanNumerals via  
70     std::ranges::copy(romanNumerals | std::views::  
71     std::cout << "\n";  
72 }
```

```
Keys in romanNumerals via std::views::elements:  
values in romanNumerals via std::views::elements
```

14.8 Intro to Parallel Algorithms

Perf  For decades, every couple of years, computer processing power

approximately doubled inexpensively. This is known as **Moore's Law**—named for Gordon Moore, co-founder of Intel and the person who identified this trend in the 1960s. Key executives at computer-processor companies NVIDIA and Arm have indicated that Moore's Law no longer applies.^{37,38} Computer processing power continues to increase, but hardware vendors now rely on new processor designs, such as **multicore processors**, which enable true parallel processing for better performance. C++ has always been focused on performance. However, its first standard support for parallelism was not added until C++11—32 years after the language's inception.

37. Esther Shein, "Moore's Law turns 55: Is it still relevant?" April 17, 2020. Accessed May 6, 2021. <https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant/>.

38. Nick Heath, "Moore's Law is dead: Three predictions about the computers of tomorrow." September 19, 2018. Accessed May 6, 2021. <https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/>.

Parallelizing an algorithm is not as simple as “flipping a switch” to say, “I want to run this algorithm in parallel.” Parallelization is sensitive to what the algorithm does and which parts can truly run in parallel on multicore hardware. Programmers must carefully determine how to divide tasks for parallel execution. Such algorithms must be scalable to any number of cores—more or fewer cores might be available at a given time because they're shared among all the computer's tasks. In addition, the number of cores is increasing over time as computer architecture evolves, so algorithms should be flexible enough to take advantage of those additional cores. As challenging as it is to write parallel algorithms, the incentive is high to maximize application performance.

17 Over the years, it has become clear that designing algorithms capable of executing on multiple cores is complex and error-prone. So many programming languages now provide built-in library capabilities that offer “canned” parallelism features. C++ already has a collection of valuable algorithms. To help programmers avoid “reinventing the wheel,” C++17 introduced **parallel overloads for 69 common-ranges algorithms**, enabling them to take advantage of multicore architectures and the high-performance “vector mathematics” operations available on today's CPUs and GPUs. Vector operations can perform the same operation on many data items at the

same time.³⁹

³⁹. “General utilities library—Execution policies—Unsequenced execution policy.” Accessed May 3, 2021. <https://eel.is/c++draft/execpol.unseq>.

17 In addition, C++17 added seven new parallel algorithms:

- `for_each_n`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `reduce`
- `transform_reduce`

23 The algorithm overloads take the burden of parallel programming largely off programmers’ shoulders and place it on the prepackaged library algorithms, leveraging the programming process. Unfortunately, the **C++20 `std::ranges` algorithms** are not yet parallelized, though they might be for C++23.⁴⁰

⁴⁰. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed May 5, 2021. <https://wg21.link/p2214r0>.

[Chapter 17](#), Concurrency, Parallelism, C++20 Coroutines and the Parallel STL will:

- overview the parallel algorithms,
- discuss the four “execution policies” (three from C++17 and one from C++20) that determine how a parallel algorithm uses a system’s parallel processing capabilities to perform a task,
- demonstrate how to invoke parallel algorithms, and
- use functions from the `<chrono>` header to time how long it takes to execute standard library algorithms running sequentially on a single core vs. running in parallel on multiple cores, so you can see the difference in performance.

20 You'll see that the parallel versions of algorithms do not always run faster than sequential versions—and we'll explain why. We'll also introduce the various standard library headers containing C++'s concurrency and parallel-programming capabilities, and introduce C++20's new coroutines feature.

14.9 Standard Library Algorithm Summary

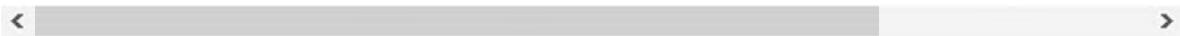
The C++ standard specifies 117 algorithms—many overloaded with two or more versions. The standard separates the algorithms into several categories:

- mutating sequence algorithms (`<algorithm>`),
- nonmodifying sequence algorithms (`<algorithm>`),
- sorting and related algorithms (`<algorithm>`),
- generalized numeric operations (`<numeric>`), and
- specialized memory operations (`<memory>`).

To learn about algorithms we did not present in this chapter, visit sites such as

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/algorithm>
<https://docs.microsoft.com/en-us/cpp/standard-lib>



Throughout this section's tables:

- Algorithms we present in this chapter are grouped at the top of each table and shown in **bold**.
- Algorithms that have a **C++20 std::ranges overload** are marked with a superscript “R.” You can see the list of **std::ranges algorithms** at:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/algorithm/ranges>



- Algorithms that have a parallel version are marked with a superscript

“P.” You can see the list of parallelized algorithms at:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/experimental/pa>

- 
- Algorithms that were added in C++ versions 11, 17 and 20 are marked with the superscript version number.

For example, the algorithm **copy** is marked with “PR,” meaning it has a parallelized version and a `std::ranges` version, and the algorithm **is_sorted_until** is marked with “PR11,” meaning it has a parallelized version and a `std::ranges` version and it was introduced to the standard library in C++11.

Mutating Sequence Algorithms

 The following table shows many of the **mutating-sequence algorithms**—i.e., algorithms that modify the containers on which they operate. The `shuffle` algorithm replaced the less-secure `random_shuffle` algorithm. “Under the hood” `random_shuffle` used function `rand`—which was inherited into C++ from the C standard library. C’s `rand` does not have “good statistical properties” and can be predictable,⁴¹ making programs that use `rand` less secure. The newer `shuffle` algorithm uses the **C++11 nondeterministic random-number generation** capabilities.

⁴¹. “Do not use the `rand()` function for generating pseudorandom numbers.” Last modified April 23, 2021. Accessed May 5, 2021.

<https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+>

Mutating sequence algorithms from header <algorithm>

<code>copy</code> ^{PR}	<code>copy_backward</code> ^R	<code>copy_if</code> ^{PR11}	<code>copy_n</code> ^{PR11}
<code>fill</code> ^{PR}	<code>fill_n</code> ^{PR}	<code>generate</code> ^{PR}	<code>generate_n</code> ^{PR}
<code>iter_swap</code>	<code>remove</code> ^{PR}	<code>remove_copy</code> ^{PR}	<code>remove_copy_if</code> ^{PR}
<code>remove_if</code> ^{PR}	<code>replace</code> ^{PR}	<code>replace_copy</code> ^{PR}	<code>replace_copy_if</code> ^{PR}
<code>replace_if</code> ^{PR}	<code>reverse</code> ^{PR}	<code>reverse_copy</code> ^{PR}	<code>shuffle</code> ^{R11}
<code>swap_ranges</code> ^{PR}	<code>transform</code> ^{PR}	<code>unique</code> ^{PR}	<code>unique_copy</code> ^{PR}
<code>move</code> ^{PR11}	<code>move_backward</code> ^{R11}	<code>rotate</code> ^{PR}	<code>rotate_copy</code> ^{PR}
<code>sample</code> ^{R17}	<code>shift_left</code> ²⁰	<code>shift_right</code> ²⁰	

Nonmodifying Sequence Algorithms

The following table shows the **nonmodifying sequence algorithms**—i.e., algorithms that do not modify the containers they operate on.

Nonmodifying sequence algorithms from header <algorithm>

<code>all_of</code> ^{PR11}	<code>any_of</code> ^{PR11}	<code>count</code> ^{PR}	<code>count_if</code> ^{PR}
<code>equal</code> ^{PR}	<code>find</code> ^{PR}	<code>find_if</code> ^{PR}	<code>find_if_not</code> ^{PR11}
<code>for_each</code> ^{PR}	<code>mismatch</code> ^{PR}	<code>none_of</code> ^{PR11}	
<code>adjacent_find</code> ^{PR}	<code>find_end</code> ^{PR}	<code>find_first_of</code> ^{PR}	<code>for_each_n</code> ^{PR17}
<code>is_permutation</code> ^{R11}	<code>search</code> ^{PR}	<code>search_n</code> ^{PR}	

Sorting and Related Algorithms

The following table shows the sorting and related algorithms.

Sorting and related algorithms from header <algorithm>

<code>binary_search</code> ^R	<code>equal_range</code> ^R	<code>includes</code> ^{PR}
<code>inplace_merge</code> ^{PR}	<code>lexicographical_compare</code> ^{PR}	<code>lower_bound</code> ^R
<code>make_heap</code> ^R	<code>max</code> ^R	<code>max_element</code> ^{PR}
<code>merge</code> ^{PR}	<code>min</code> ^R	<code>min_element</code> ^{PR}
<code>minmax</code> ^{R11}	<code>minmax_element</code> ^{PR11}	<code>pop_heap</code> ^R
<code>push_heap</code> ^R	<code>set_difference</code> ^{PR}	<code>set_intersection</code> ^{PR}
<code>set_symmetric_difference</code> ^{PR}	<code>set_union</code> ^{PR}	<code>sort</code> ^{PR}
<code>sort_heap</code> ^R	<code>upper_bound</code> ^R	
<code>clamp</code> ^{R17}	<code>is_heap</code> ^{PR11}	<code>is_heap_until</code> ^{PR11}
<code>is_partitioned</code> ^{PR11}	<code>is_sorted</code> ^{PR11}	<code>is_sorted_until</code> ^{PR11}
<code>lexicographical_compare_three_way</code> ²⁰		<code>next_permutation</code> ^R
<code>nth_element</code> ^{PR}	<code>partial_sort</code> ^{PR}	<code>partial_sort_copy</code> ^{PR}
<code>partition</code> ^{PR}	<code>partition_copy</code> ^{PR11}	<code>partition_point</code> ^{R11}
<code>prev_permutation</code> ^R	<code>stable_partition</code> ^{PR}	<code>stable_sort</code> ^{PR}

Numerical Algorithms

23 The following table shows the numerical algorithms of the header <`numeric`>. The algorithms in this header have not yet been updated for C++20 ranges, though they are being worked on for inclusion in C++23.^{42,43}

42. Christopher Di Bella, “A Concept Design for the Numeric,” August 2, 2019. Accessed May 2, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1813r0.pdf>.

43. Tristan Brindle, “Numeric Range Algorithms for C++20,” May 19, 2020. Accessed May 2, 2021. <https://tristanbrindle.com/posts/numeric-ranges-for-cpp20>.

Generalized numeric operations from header <numeric>

accumulate	gcd ¹⁷	iota ¹¹
lcm ¹⁷	partial_sum	reduce ¹⁷
adjacent_difference ^P	exclusive_scan ^{P17}	inclusive_scan ^{P17}
inner_product ^P	midpoint ²⁰	transform_reduce ^{P17}
transform_exclusive_scan ^{P17}	transform_inclusive_scan ^{P17}	

Specialized Memory Operations

The following table shows the specialized memory algorithms of the header <**memory**>, which contains features for dynamic memory management that are beyond this book's scope. For an overview of the header and these algorithms, see:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/header/memory>

Specialized Memory Operations

construct_at ^{R20}	destroy ^{R17}
destroy_at ^{R17}	destroy_n ^{R17}
uninitialized_copy ^{PR}	uninitialized_copy_n ^{PR11}
uninitialized_default_construct ^{R17}	uninitialized_default_construct_n ^{R17}
uninitialized_fill ^{PR}	uninitialized_fill_n ^{PR}
uninitialized_move ^{R17}	uninitialized_move_n ^{R17}
uninitialized_value_construct ^{R17}	uninitialized_value_construct_n ^{R17}

23 14.10 A Look Ahead to C++23 Ranges

Though many algorithms have overloads in the **C++20 std::ranges namespace**, various C++20 algorithms—including those in the <numeric>

header and the parallel algorithms in the `<algorithm>` header—do not yet have C++20 `std::ranges` overloads. There are standard committee proposals for various additional ranges library features that might be part of C++23. Some of these capabilities are under development and can be used now via the open-source project `rangesnext`.⁴⁴ C++20’s ranges functionality and many of the new features proposed for C++23 are based on capabilities found in the open-source project `range-v3`.⁴⁵

44. Corentin Jabot, “Ranges For C++23.” Accessed May 5, 2021. <https://github.com/cor3ntin/rangesnext>.

45. Eric Niebler, “range-v3.” Accessed May 5, 2021. <https://github.com/ericniebler/range-v3>.

“A Plan for C++23 Ranges”⁴⁶

46. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed May 5, 2021. <https://wg21.link/p2214r0>.

This paper provides an overview of the general plan for C++23 `ranges`, plus details on many possible new ranges features. The proposed features are categorized by importance into three tiers, with Tier 1 containing the most important features. After a brief introduction, the paper discusses four categories of possible additions:

- **View adjuncts:** This section briefly discusses two key features—the overloaded `ranges::to` function for converting views to various container types and the ability to format views and ranges for convenient output. These are discussed in detail in the papers “`ranges::to`: A function to convert any range to a container”⁴⁷ and “Formatting Ranges,”⁴⁸ respectively.

47. Corentin Jabot, Eric Niebler and Casey Carter, “`ranges::to`: A function to convert any range to a container,” November 22, 2021. Accessed May 5, 2021. <https://wg21.link/p1206r3>.

48. Barry Revzin, “Formatting Ranges,” February 19, 2021. Accessed May 5, 2021. <https://wg21.link/p2286>.

- **Algorithms:** This section overviews potential new `std::ranges` overloads for various `<numeric>` algorithms, which are discussed in more detail in the paper “A Concept Design for the Numeric Algorithms.”⁴⁹ This section also briefly overviews potential issues with

producing `std::ranges` overloads of C++17's parallel algorithms. The paper "Introduce Parallelism to the Ranges TS" provides more in-depth discussions of these issues.

49. Christopher Di Bella, "A Concept Design for the Numeric Algorithms," August 2, 2019. Accessed May 5, 2021. <http://wg21.link/p1813r0>.

- **Actions:** These are a third category of capabilities separate from ranges and views in the **range-v3 project**. Actions, like **views**, are composable with the **| operator**, but, like the **std::ranges algorithms**, actions are **greedy**, so they immediately produce results. According to this section, though actions would make some coding more convenient, adding them is a low priority because you can perform the same tasks by calling existing `std::ranges` algorithms in multiple statements.

14.11 Wrap-Up

In this chapter, we demonstrated many of the standard library algorithms, including filling containers with values, generating values, comparing elements or entire containers, removing elements, replacing elements, mathematical operations, searching, sorting, swapping, copying, merging, set operations, determining boundaries, and calculating minimums and maximums. We focused primarily on the C++20 `std::ranges` versions of these algorithms.

You saw that the standard library's algorithms specify various minimum requirements that help you determine which containers, iterators and functions can be passed to each algorithm. We overviewed some named requirements used by the common-ranges algorithms, then indicated that the C++20 range-based algorithms use C++20 concepts to specify their requirements, which are checked at compile time. We briefly introduced the C++20 concepts specified for each range-based algorithm we presented. Not all algorithms have a range-based version.

We revisited lambdas and introduced additional capabilities for capturing the enclosing scope's variables. You saw that many algorithms can receive a lambda, a function pointer or a function object as an argument, and call them to customize the algorithms' behaviors.

We continued our discussion of C++'s functional-style programming. We showed how to create a logically infinite sequence of values and how to use

range adaptors to limit the total number of elements processed through a pipeline. We saved views in variables for later use and added more steps to a previously saved pipeline. We introduced range adaptors for manipulating the keys and values in key-value pairs, and showed a similar range adaptor for selecting any indexed element from fixed-size objects, like `pairs`, `tuples` and `arrays`.

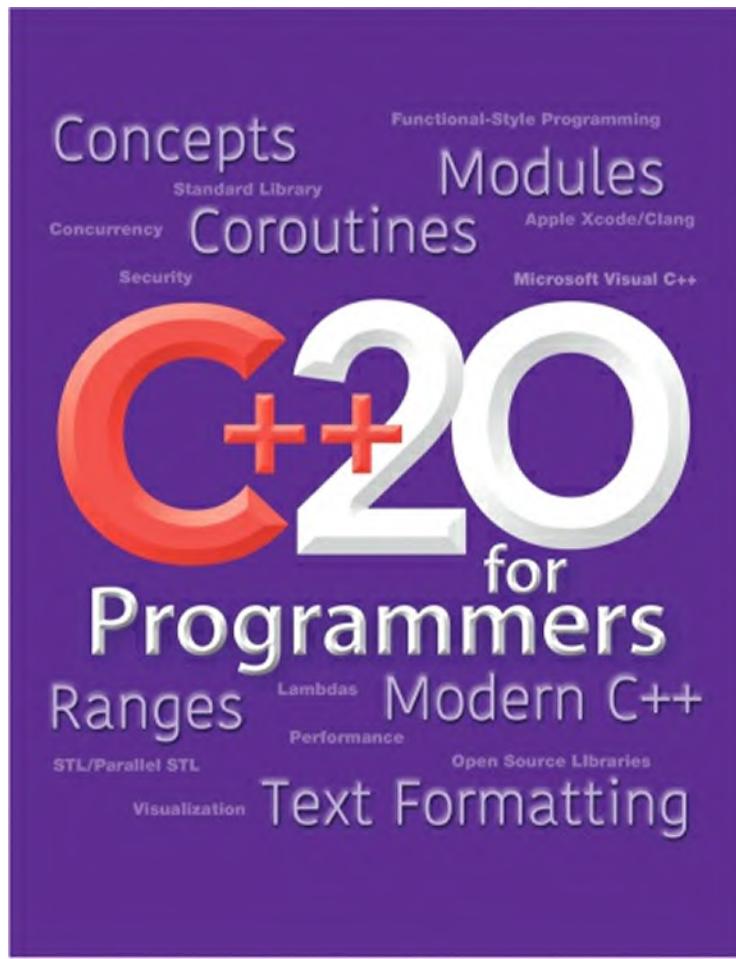
You learned that C++17 introduced new parallel overloads for 69 standard library algorithms in the `<algorithm>` header. As you'll see in [Chapter 16](#), these will enable you to take advantage of your computer's multi-core hardware to enhance program performance. In [Chapter 16](#), we'll demonstrate several parallel algorithms and use the `<chrono>` header's capabilities to time sequential and parallel algorithm calls, so you can see the performance differences. We'll explain why parallel algorithms do not always run faster than their sequential counterparts, so it's not always worthwhile to use the parallel versions.

You saw that various C++20 algorithms, including those in the `<numeric>` header and the parallel algorithms in the `<algorithm>` header, do not have `std::ranges` overloads. We mentioned the updates expected in C++23 and pointed you to the GitHub project `rangesnext`, which contains implementations for many of the proposed updates.

In the next chapter, we'll build a simple container, simple iterator and a simple algorithm using custom templates, using C++20 concepts in our templates as appropriate.

Part 5: Advanced Topics

Chapter 15. Templates, C++20 Concepts and Metaprogramming



Objectives

In this chapter, you'll:

- Appreciate the rising importance of generic programming.
- Use class templates to create related custom classes.
- Understand compile-time vs. runtime polymorphism.
- Distinguish between templates and template specializations.
- Use C++20 abbreviated function templates and templated lambdas.

- Use C++20 concepts to constrain template parameters and to overload function templates based on their type requirements.
 - Use type traits and see how they relate to C++20 concepts.
 - Test concepts at compile-time with `static_assert`.
 - Create a custom concept-constrained algorithm.
 - Rebuild our class `MyArray` as a custom container class template with custom iterators.
 - Use non-type template parameters to pass compile-time constants to templates, and use default template arguments.
 - Use variadic templates that receive any number of parameters and apply binary operators to them via fold expressions.
 - Use compile-time template metaprogramming capabilities to compute values, manipulate types and generate code to improve runtime performance.
-

Outline

15.1 Introduction

15.2 Custom Class Templates and Compile-Time Polymorphism

15.3 C++20 Function Template Enhancements

15.3.1 C++20 Abbreviated Function Templates

15.3.2 C++20 Templated Lambdas

15.4 C++20 Concepts: A First Look

15.4.1 Function Template `multiply`

15.4.2 Constrained Function Template with a C++20 Concepts requires Clause

15.4.3 C++20 Predefined Concepts

15.5 Type Traits

15.6 C++20 Concepts: A Deeper Look

15.6.1 Creating a Custom Concept

15.6.2 Using a Concept

15.6.3 Using Concepts in Abbreviated Function Templates

- 15.6.4 Concept-Based Overloading**
 - 15.6.5 `requires` Expressions**
 - 15.6.6 C++20 Exposition-Only Concepts**
 - 15.6.7 Techniques Before C++20 Concepts: SFINAE and Tag Dispatch**
 - 15.7 Testing C++20 Concepts with `static_assert`**
 - 15.8 Creating a Custom Algorithm**
 - 15.9 Creating a Custom Container and Iterators**
 - 15.9.1 Class Template `ConstIterator`**
 - 15.9.2 Class Template `Iterator`**
 - 15.9.3 Class Template `MyArray`**
 - 15.9.4 `MyArray` Deduction Guide for Braced Initialization**
 - 15.9.5 Using `MyArray` and Its Custom Iterators with `std::ranges` Algorithms**
 - 15.10 Default Arguments for Template Type Parameters**
 - 15.11 Variable Templates**
 - 15.12 Variadic Templates and Fold Expressions**
 - 15.12.1 `tuple` Variadic Class Template**
 - 15.12.2 Variadic Function Templates and an Intro to C++17 Fold Expressions**
 - 15.12.3 Types of Fold Expressions**
 - 15.12.4 How Unary Fold Expressions Apply Their Operators**
 - 15.12.5 How Binary-Fold Expressions Apply Their Operators**
 - 15.12.6 Using the Comma Operator to Repeatedly Perform an Operation**
 - 15.12.7 Constraining Parameter Pack Elements to the Same Type**
 - 15.13 Template Metaprogramming**
 - 15.13.1 C++ Templates Are Turing Complete**
 - 15.13.2 Computing Values at Compile Time**
 - 15.13.3 Conditional Compilation with Template Metaprogramming and `constexpr if`**
 - 15.13.4 Type Metafunctions**
 - 15.14 Wrap-Up**
-

15.1 Introduction

 Generic programming with templates has been in C++ since the 1998 C++ standard was released,¹ and has risen in importance with each new C++ release. A modern C++ theme is to do more at compile time for better type-checking and better runtime performance.^{2,3,4} You've already used templates extensively. As you'll see, templates and especially template metaprogramming are the keys to powerful compile-time operations. In this chapter, we'll take a deeper look templates, as we explain how to develop custom class templates, explore concepts—C++20's most significant new feature—and introduce template metaprogramming. The following table summarizes the book's templates coverage across all 19 chapters.

1. “History of C++.” Accessed June 24, 2021. <https://www.cplusplus.com/info/history/>.
2. “C++ Core Guidelines—Per: Performance.” Accessed February 8, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance>.
3. “Big Picture Issues—What’s the big deal with generic programming?” Accessed February 8, 2021. <https://isocpp.org/wiki/faq/big-picture#generic-paradigm>.
4. “Compile time vs run time polymorphism in C++ advantages/disadvantages.” Accessed January 31, 2021. <https://stackoverflow.com/questions/16875989/compile-time-vs-run-time-polymorphism-in-c-advantages-disadvantages>.

Chapter	Templates coverage
Chapter 1	Introduction to generic programming.
Chapters 2–4	<code>strings</code> (which are class templates).
Chapter 5	Section 5.10: <code>uniform_int_distribution</code> class template for random-number generation. Section 5.17: Defining a function template.
Chapter 6	Standard library container class templates <code>array</code> and <code>vector</code> .
Chapter 7	Section 7.10: Class template <code>span</code> for creating a view into a container of contiguous elements, such as an <code>array</code> or <code>vector</code> .
Chapter 8	Sections 8.2–8.9: In-depth treatment of <code>strings</code> (which are class templates). Sections 8.12–8.16: File-stream-processing classes (which are class templates). Section 8.17: <code>string-stream</code> class templates. Section 8.19: <code>rapidcsv</code> library function templates for manipulating CSV data.
Chapter 9	Section 9.22: <code>cereal</code> library function templates for serializing and deserializing data using JavaScript Object Notation (JSON).
Chapter 10	Section 10.13: Runtime polymorphism with the <code>std::variant</code> class template and the <code>std::visit</code> function template.
Chapter 11	Smart pointers for managing dynamically allocated memory with class template <code>unique_ptr</code> , function template <code>make_unique</code> , and class template <code>initializer_list</code> for passing initializer lists to functions.
Chapter 12	Section 12.8: <code>unique_ptr</code> class template.
Chapter 13	Standard library container class templates and iterators (which also are implemented as class templates).
Chapter 14	Standard library algorithm function templates that manipulate standard library container class templates via iterators.
Chapter 15	Custom class templates, iterator templates, function templates, abbreviated function templates, templated lambdas, type traits, C++20 concepts, concept-based overloading, variable templates, alias templates, variadic templates, fold expressions and template metaprogramming.
Chapter 16	Parallel standard library algorithms and various other function templates and class templates related to multithreaded application development.
Chapter 18	(Online) Stream I/O classes (which are class templates).
Chapter 19	(Online) Section 19.15: <code>shared_ptr</code> and <code>weak_ptr</code> smart-pointer class templates.

Custom Templates

Section 5.17 showed that the compiler uses function templates to generate overloaded functions. The generated functions are called **function-template specializations**. Similarly, the compiler uses class templates to generate related classes called **class-template specializations**. Specializing templates enables **compile-time (static) polymorphism**. In this chapter, you'll create custom class templates and study key template-related technologies.

C++20 Template Features

20 Concepts  We discuss C++20's new template capabilities, including **abbreviated function templates**, **templated lambdas** and **concepts**—a C++20 “big four” feature that makes generic programming with templates even more convenient and powerful. When invoking C++20's **range-based algorithms**, you saw that you must pass container or iterator arguments that meet the algorithms' requirements. This chapter takes a template developer's viewpoint as we develop custom templates that **specify their requirements** using predefined and custom C++20 concepts.^{5,6,7,8} The compiler checks concepts *before* specializing templates, often resulting in fewer, clearer and more-precise error messages.

5. Marius Bancila. “Concepts versus SFINAE-based constraints.” Accessed February 4, 2021. <https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-based-constraints/>.
6. Saar Raz. “C++20 Concepts: A Day in the Life,” YouTube video, October 17, 2019, <https://www.youtube.com/watch?v=qawSiMIXtE4>.
7. “Constraints and concepts.” Accessed February 4, 2021. <https://en.cppreference.com/w/cpp/language/constraints>.
8. “Concepts library.” Accessed February 4, 2021. <https://en.cppreference.com/w/cpp/concepts>.

Type Traits

We'll introduce **type traits** for testing attributes of built-in and custom class types at compile time. You'll see that many C++20 concepts are implemented in terms of type traits. Concepts simplify using type traits to constrain template parameters.

Building Custom Containers, Iterators and Algorithms

[Chapter 11](#)'s MyArray class stored only int values. We'll create a MyArray class template that can be specialized to store elements of various types (e.g., MyArray of float or MyArray of Employee). We'll make MyArray objects compatible with many standard library algorithms by defining **custom iterators**. We'll also define a **custom algorithm** that can process elements of MyArrays and standard library container class objects.

Variadic Templates and Fold Expressions

17 We'll build a **variadic function template** that receives one or more parameters. We'll also introduce **C++17 fold expressions**, which enable you to conveniently apply an operation to all the items passed to a variadic template.

Template Metaprogramming

Perf  The C++ Core Guidelines define template metaprogramming (TMP) as “**creating programs that write code at compile time**.”⁹ The compiler also can use them to perform **compile-time calculations** and **type manipulations**. Compile-time calculations enable you to improve a program’s execution-time performance, possibly reducing execution time and memory consumption. You’ll see that **type traits** are used extensively in template metaprogramming for generating code based on template-argument attributes. We’ll also write a function template that generates different code at compile-time, based on whether its container argument supports **random-access iterators** or **lesser iterators**. You’ll see how this enables us to **optimize the program’s runtime performance**.

⁹. “C++ Core Guidelines—T: Templates and generic programming.” Accessed June 24, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates>.

20 CG  Before C++20, many C++ programmers viewed template metaprogramming as too complex to use. C++20 concepts make aspects of it friendlier and more accessible. Google’s C++ Style Guide nevertheless says to “avoid complicated template programming.”¹⁰ Similarly, the C++ Core Guidelines indicate that you should “use template metaprogramming only when you really need to” and that it’s “hard to get right, ... and is often hard to maintain.”¹¹ These guidelines could be updated when developers gain

more experience using C++20 concepts and enhancements possibly coming in C++23.

10. “Google C++ Style Guide—Template Metaprogramming.” Accessed June 27, 2021.
https://google.github.io/styleguide/cppguide.html#Template_metapro
11. “T.120: Use template metaprogramming only when you really need to.” Accessed June 27, 2021.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta>.

15.2 Custom Class Templates and Compile-Time Polymorphism

A template enables **compile-time polymorphism** (also called **static polymorphism**)^{12,13} by specifying capabilities generically, then letting the compiler specialize the template, generating type-specific code versions on demand.

12. “C++ — Static Polymorphism.” Accessed January 31, 2021.
https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.

Class templates are called **parameterized types** because they require one or more **type parameters** to tell the compiler how to customize a generic class template to form a **class-template specialization** from which objects can be instantiated. You write one class-template definition. **When you need particular specializations, you use concise, simple notations, and the compiler writes only those specializations for you.** One Stack class template, for example, could become the basis for creating many Stack class-template specializations, such as “Stack of doubles,” “Stack of ints,” “Stack of Employees,” “Stack of Bills,” or “Stack of ActivationRecords.”

To use a type with a template, the type must meet the template’s requirements. For example, a template might require objects of a specified type to

- be **initializable with a default constructor**,
- be **copyable** or **movable**,
- be **comparable with operator <** to determine their sort order,
- have **specific member functions**, and more.

 **Compilation errors occur when you specialize a template with a type that does not meet the template's requirements.**

Creating Class Template Stack<T>

 Let's jump into coding a custom Stack class template. It's possible to understand the notion of a stack **independently of the kinds of items you place onto or remove from it**—stacks are simply **last-in, first-out (LIFO)** data structures. Class templates encourage software reusability by enabling the compiler to instantiate many type-specific class-template specializations from a single class template—as you saw with the **stack container adapter** in Section 13.10.1. Here (Fig. 15.1), we define a stack generically then instantiate and use type-specific stacks (Fig. 15.2). Fig. 15.1's Stack class-template definition looks like a conventional class definition, with a few key differences.

13. Bondarenko, Kateryna. "Static Polymorphism in C++." May 6, 2019. Accessed January 31, 2021. <https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ae27a945b>.

[Click here to view code image](#)

```
1 // Fig. 15.1: Stack.h
2 // Stack class template.
3 #pragma once
4 #include <deque>
5
6 template<typename T>
7 class Stack {
8 public:
9     // return the top element of the Stack
10    const T& top() {
11        return stack.front();
12    }
13
14    // push an element onto the Stack
15    void push(const T& pushValue) {
16        stack.push_front(pushValue);
```

```

17      }
18
19      // pop an element from the stack
20      void pop() {
21          stack.pop_front();
22      }
23
24      // determine whether Stack is empty
25      bool isEmpty() const {
26          return stack.empty();
27      }
28
29      // return size of Stack
30      size_t size() const {
31          return stack.size();
32      }
33
34  private:
35      std::deque<T> stack{}; // internal repre
36  };

```



Fig. 15.1 | Stack class template.

template Header

The first key difference is the **template header** (line 6):

```
template<typename T>
```

which begins with the **template** keyword, followed by a comma-separated list of **template parameters** enclosed in **angle brackets** (< and >). Each template parameter that represents a type must be preceded by one of the interchangeable keywords **typename** or **class**. Some programmers prefer **typename** because a template's **type arguments** might not be class types.¹⁴ The **type parameter** **T** acts as a **placeholder** for the **Stack's element type**. Type parameter names can be any valid identifier but must be unique inside a template definition. The element type **T** is mentioned throughout the **Stack**

class template wherever we need access to the `Stack`'s element type:

14. Scott Meyers, “Item 42: Understand the Meanings of `typename`,” *Effective C++*, *Third Edition*. Pearson Education, Inc., 2005.

- declaring a member function return type (line 10),
- declaring a member function parameter (line 15), and
- declaring a variable (line 35).

The compiler associates the **type parameter** with a **type argument** when you specialize the class template. At that point, the compiler generates a copy of the class template in which all occurrences of the **type parameter** are replaced with the specified type. **Compilers generate definitions only for the portions of a template used in your code.**^{15 16}

15. Andreas Fertig, “Back to Basics: Templates (Part 1 of 2),” September 25, 2020. Accessed June 14, 2020. <https://www.youtube.com/watch?v=VNJ4wiuxJM4>.

16. “13.9.2 Implicit instantiation [temp.inst].” Accessed June 20, 2021. <https://eel.is/c++draft/temp.inst#4>.

SE A Another key difference between class template `Stack` and other classes we've defined is that **we did not separate the class template's interface from its implementation**. You define templates in headers, then `#include` them in client-code files. **The compiler needs the full template definition each time the template is specialized with new type arguments, so it can generate the appropriate code.** For class templates, this means that the member functions also are defined in the header—typically inside the class definition's body, as in Fig. 15.1.

Class Template `Stack<T>`'s Data Representation

Perf  Section 13.10.1 showed that the standard library's **stack adapter class** can use various containers to store its elements. A **stack** requires insertions and deletions only at its **top**, so a **vector** or a **deque** could be used to store the **stack**'s elements. A **vector** supports fast insertions and deletions at its **back**. A **deque** supports fast insertions and deletions at its **front** and its **back**. A **deque** is the default representation for the standard library's **stack adapter**¹⁷ because a **deque grows more efficiently than a vector**:

- A `vector`’s elements are stored in a **contiguous block of memory**. When that block is full and you add a new element, the `vector` performs the expensive operation of allocating a larger **contiguous block of memory** and **copying** the old elements into that new block.
- A `deque`, on the other hand, is typically implemented as a **list of fixed-size, builtin arrays**—with new ones added as necessary. **No existing elements are copied when new items are added to a deque’s front or back.**

For these reasons, we use a `deque` (line 35) as our `Stack` class’s underlying container.

Class Template `Stack<T>`’s Member Function Templates

A class template’s member-function definitions are function templates. When you define them within the class template’s body, you do **not precede them with a template header**. However, they do use the **template parameter T** to represent the element type. Our `Stack` class template does not define its own constructors—the class’s default constructor invokes the **deque data member’s default constructor**.

Our `Stack` class (Fig. 15.1) provides the following member functions:

- `top` (lines 10–12) returns a `const` reference to the `Stack`’s **top** element—it does not remove that element from the stack.
- `push` (lines 15–17) places a new element on the **top** of the `Stack`.
- `pop` (lines 20–22) removes the `Stack`’s **top** element.
- `isEmpty` (lines 25–27) returns the `bool` value `true` if the `Stack` is **empty**; otherwise, it returns `false`.
- `size` (lines 30–32) returns the number of elements in the `Stack`.

Each calls a `deque` member function to perform its task—this is known as **delegation**.

Testing Class Template `Stack<T>`

Figure 15.2 tests the `Stack` class template. Line 8 instantiates

doubleStack. This variable is declared as type **Stack<double>**—pronounced “Stack of double.” The compiler associates type **double** with **type parameter T** in the class template to produce the source code for a **Stack** class with **double** elements that stores its elements in a **deque<double>**. Lines 15–19 invoke **push** (line 16) to place the **double** values 1.1, 2.2, 3.3, 4.4 and 5.5 onto **doubleStack**. Next, lines 24–27 invoke **isEmpty**, **top** and **pop** in a while loop to remove all of the stack’s elements. Notice in the output of Fig. 15.2 that the values indeed **pop** off in **last-in, first-out order**. When **doubleStack** is empty, the **pop** loop terminates.

[Click here to view code image](#)

```
1 // fig15_02.cpp
2 // Stack class template test program.
3 #include <iostream>
4 #include "Stack.h" // Stack class template
5 using namespace std;
6
7 int main() {
8     Stack<double> doubleStack{}; // create a
9     constexpr size_t doubleStackSize{5}; // .
10    double doubleValue{1.1}; // first value
11
12    cout << "Pushing elements onto doubleSta
13
14    // push 5 doubles onto doubleStack
15    for (size_t i{0}; i < doubleStackSize; +
16        doubleStack.push(doubleValue);
17        cout << doubleValue << ' ';
18        doubleValue += 1.1;
19    }
20
21    cout << "\n\nPopping elements from doubl
22
23    // pop elements from doubleStack
```

```
24     while (!doubleStack.isEmpty()) { // loop
25         cout << doubleStack.top() << ' '; // dis-
26         doubleStack.pop(); // remove top elem-
27     }
28
29     cout << "\nStack is empty, cannot pop.\n"
30
31     Stack<int> intStack{}; // create a Stack
32     constexpr size_t intStackSize{10}; // st-
33     int intValue{1}; // first value to push
34
35     cout << "\nPushing elements onto intStack"
36
37     // push 10 integers onto intStack
38     for (size_t i{0}; i < intStackSize; ++i)
39         intStack.push(intValue);
40         cout << intValue++ << ' ';
41     }
42
43     cout << "\n\nPopping elements from intStack"
44
45     // pop elements from intStack
46     while (!intStack.isEmpty()) { // loop wh-
47         cout << intStack.top() << ' '; // dis-
48         intStack.pop(); // remove top element
49     }
50
51     cout << "\nStack is empty, cannot pop.\n"
52 }
```

< >

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop.

```
Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop.
```

Fig. 15.2 | Stack class template test program. (Part 2 of 2.)

Line 31 instantiates `intStack`. This variable is declared as type `Stack<int>`—pronounced “Stack of int.” Lines 38–41 repeatedly invoke `push` (line 39) to place values onto `intStack`, then lines 46–49 repeatedly invoke `isEmpty`, `top` and `pop` to remove values from `intStack` until it’s empty. Once again, the output confirms the **last-in, first-out order** in which the elements are removed.

Though the compiler does not show you the generated code for `Stack<double>` and `Stack<int>`, you can see sample generated code by using the website:

<https://cppinsights.io>

This site shows the template specializations generated by the Clang compiler. (The site requires all the code to be pasted into its code pane. To try this with our example, you’ll need to remove the `#include` for `stack.h` in the `main` program and paste class template `Stack`’s code above `main`.)

Defining Class Template Member Functions Outside a Class Template

Member-function definitions can be defined outside a **class template definition**. In this case, each member-function definition must begin with the same **template header** as the class template. Also, you must **qualify each member function with the class name and scope resolution operator**. For example, you’d define the `pop` function outside the class-template definition as:

[Click here to view code image](#)

```
template<typename T>
```

```
inline void Stack<T>::pop() {
    stack.pop_front();
}
```

 **Stack<T>::** indicates that **pop** is in class **template Stack<T>**'s scope. The standard library generally defines all container-class member functions inside their class definitions.

20 15.3 C++20 Function Template Enhancements

In addition to concepts, C++20 added **abbreviated function templates** and **templated lambda expressions**.

20 15.3.1 C++20 Abbreviated Function Templates

Using traditional function-template syntax, we could define a `printContainer` function template with a template header as follows:

[Click here to view code image](#)

```
template <typename T>
void printContainer(const T& items) {
    for (const auto& item : items) {
        std::cout << item << " ";
    }
}
```

20 The function template receives a reference to a container (`items`) and uses a range-based `for` statement to display the elements. C++20 **abbreviated function templates** (Fig. 15.3) enable you to **define a function template without the template header** (lines 10–14). Line 10 uses the **auto keyword** to infer the parameter type.

[Click here to view code image](#)

```
1 // fig15_03.cpp
2 // Abbreviated function template.
```

```

3 #include <array>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 // abbreviated function template printConta
9 // container, such as an array or a vector
10 void printContainer(const auto& items) {
11     for (const auto& item : items) {
12         std::cout << item << " ";
13     }
14 }
15
16 int main() {
17     using namespace std::string_literals; //
18
19     std::array ints{1, 2, 3, 4, 5};
20     std::vector strings{"red"s, "green"s, "b
21
22     std::cout << "ints: ";
23     printContainer(ints);
24     std::cout << "\nstrings: ";
25     printContainer(strings);
26     std::cout << "\n";
27 }
```

< >

```
ints: 1 2 3 4 5
strings: red green blue
```

Fig. 15.3 | Abbreviated function template.

Recall that a string literal like "red" has the type `const char*`. Rather than string literals, line 20 uses **string object literals** denoted by the `s` that follows each string literal's closing quote. This enables the compiler to **deduce from the initializer list** that the `vector`'s element type is

`std::string` rather than `const char*`. To use **string object literals**, include the `<string>` header. Line 17's specifies that the program uses namespace `std::string_literals`, which contains features that support **string object literals**.

Sometimes Traditional Function Template Syntax Is Required

The abbreviated function template syntax is similar to that of regular function definitions, but **is not always appropriate**. Consider the first two lines of Section 5.17's function template `maximum`, which received three parameters of the **same type**:

[Click here to view code image](#)

```
template <typename T>
T maximum(T value1, T value2, T value3) {
```

SE A These lines show the correct way to **ensure that all three parameters have the same type**.

If we write `maximum` as an **abbreviated function template**:

[Click here to view code image](#)

```
auto maximum(auto value1, auto value2, auto value3)
```

the compiler independently infers each `auto` parameter's type based on the corresponding argument. So, `maximum` could receive arguments of three different types.

Specializing `printContainer` with an Incompatible Type

Err X When the compiler attempts to specialize `printContainer`, errors will occur if:

- we pass an **object that is incompatible with a range-based `for` statement** or
- objects of the **container's element type cannot be output with the `<<` operator**,

In Sections 15.4 and 15.6, we'll show how **C++20 concepts can be used to constrain the types passed to a function template and prevent the**

compiler from attempting to specialize the template.

20 15.3.2 C++20 Templated Lambdas

In C++20, lambdas can specify template parameters. Consider the following lambda from Fig. 14.16, which we used to calculate the sum of the squares of an array's integers:

[Click here to view code image](#)

```
[ ](auto total, auto value) {return total + value}
```

 A horizontal bar with arrows at both ends, indicating scrollable content.

SE  The compiler independently infers `total`'s and `value`'s (possibly different) types from their arguments. **You can force the lambda to require the same type for both** by using a **templated lambda**:

[Click here to view code image](#)

```
[ ]<typename T>(T total, T value) {return total +
```

 A horizontal bar with arrows at both ends, indicating scrollable content.

Err  The template parameter list is placed between the **lambda's introducer** and lambda's parameter list. **When you use one template type parameter (T) to declare both lambda parameters, the compiler requires both arguments to have the same type**; otherwise, a compilation error occurs.

20 15.4 C++20 Concepts: A First Look

Concepts  **Concepts** (briefly introduced in [Section 14.2](#)) simplify C++ template metaprogramming. C++ experts say that “Concepts are a revolutionary approach for writing templates”¹⁸ and that “C++20 creates a paradigm shift in the way we use metaprogramming.”¹⁹ C++’s creator, Bjarne Stroustrup, says, “Concepts complete C++ templates as originally envisioned” and that they’ll “dramatically improve your generic programming and make the current workarounds (e.g., traits classes) and low-level techniques (e.g., `enable_if`-based overloading) feel like error-

prone and tedious assembly programming.”²⁰

18. Bartłomiej Filipek, “C++20 Concepts — a Quick Introduction,” May 5 2021. Accessed May 29, 2021. <https://www.cppstories.com/2021/concepts-intro/>.
19. Inbal Levi, “Exploration of C++20 Meta Programming,” September 29, 2020. Accessed May 29, 2021. <https://www.youtube.com/watch?v=XgrjybKaIV8>.
20. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying template interfaces,” January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.

As you’ll see, concepts explicitly constrain the arguments specified for a template’s parameters. You’ll use **requires clauses** and **requires expressions** to specify constraints, which can

- **test attributes of types**—such as, “Is the type an integer type?”—and
- **test whether types support various operations**—such as, “Does a type support the comparison operations?”

The C++ standard provides **74 predefined concepts** (see [Section 15.4.3](#)) and you can create custom concepts. Each concept defines a type’s requirements or a relationship between types.²¹ **Concepts can be applied to any parameter of any template and to any use of auto.**²² We’ll discuss concepts in more depth in [Section 15.6](#). Initially, we’ll focus on using concepts to restrict a function template’s type parameters.

21. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying template interfaces,” January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.
22. “Placeholder type specifiers.” Accessed June 14, 2021. <https://en.cppreference.com/w/cpp/language/auto>.

Motivation and Goals of Concepts

Stroustrup says, “Concepts enable overloading and eliminate the need for a lot of ad-hoc metaprogramming and much metaprogramming scaffolding code, thus significantly simplifying metaprogramming as well as generic programming.”²³

23. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3 Using Concepts,” January 31, 2017. Accessed June 14, 2021. <http://wg21.link/p0557r0>.

Traditionally, template requirements were **implicit** in how the template used its arguments in operator expressions, function calls, etc.²⁴ Though such requirements typically would be documented in program comments, **the compiler cannot enforce comments**. To determine that a type was

incompatible with a template, the compiler first had to attempt a specialization to “see” that a type did not support the template’s implicit requirements. This often lead to significant (if not vast) numbers of compilation errors.

24. Hendrik Niemeyer, “An Introduction to C++20’s Concepts,” July 25, 2020. Accessed June 1, 2021. https://www.youtube.com/watch?v=N_kPd2OK1I8.

SE A Concepts specify template requirements explicitly in code (e.g., in a function template’s signature or a class template’s template header). They enable the compiler to determine that a type is not compatible with a template *before* specializing it—leading to fewer, more precise error messages.²⁵

25. The error-messages’ quality and quantity still varies considerably among compilers.

SE A Concepts also enable you to overload function templates with the same signature based on each function template’s requirements. For example, we’ll define two different overloads of a function template with the same signature:

- one will support any container with input iterators, and
- the other will be optimized for containers with random-access iterators.

15.4.1 Unconstrained Function Template multiply

When you call a function, the compiler uses its **overload-resolution rules**²⁶ and a technique called **argument-dependent lookup (ADL)**^{27,28,29} to locate all the function definitions—known as the **overload set**—that might satisfy the function call. This process often includes **specializing function templates** based on a function call’s argument types. The compiler then chooses the best match from the overload set. **An unconstrained function template does not explicitly specify any requirements.** So, the compiler must attempt to specialize the template to check if the function call’s arguments support the operations in that template’s definition.

26. “Overload resolution.” Accessed May 30, 2021. https://en.cppreference.com/w/cpp/language/overload_resolution.

27. “Working Draft, Standard for Programming Language C++—6.5.4 Argument-dependent name

- lookup [basic.lookup.argdep]," Accessed June 18, 2021.
<https://eel.is/c++draft/basic.lookup.argdep#:lookup,argument-dependent>.
28. “Argument-dependent lookup.” June 14, 2021.
<https://en.cppreference.com/w/cpp/language/adl>.
29. Inbal Levi. “Exploration of C++20 Metaprogramming.” September 29, 2020. Accessed June 14, 2021. <https://www.youtube.com/watch?v=XgrjybKaIV8>.

Consider the **unconstrained function template** multiply (lines 5–8 of Fig. 15.4), which receives **two values of the same type** (T) and returns their product.

[Click here to view code image](#)

```
1 // fig15_04.cpp
2 // Simple unconstrained multiply function template
3 #include <iostream>
4
5 template<typename T>
6 T multiply(T first, T second) {
7     return first * second;
8 }
9
10 int main() {
11     std::cout << "Product of 5 and 3: " << m
12             << "\nProduct of 7.25 and 2.0: " << m
13 }
```

```
< >
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
```

Fig. 15.4 | Simple unconstrained multiply function template.

This template has **implicit requirements** that you can infer from the code:

- The parameter types are not pointers or references, so the **arguments are received by value**. Similarly, the return type is not a pointer or a

reference, so the result is returned by value (line 6). So, **the arguments' type must support copying**.

- The arguments are multiplied (line 7), so **the arguments' type must support the binary * operator**—either natively (as with built-in types like `int` and `double`) or via operator overloading.

When the compiler specializes `multiply` for a given type and determines that the arguments are not compatible with the template's implicit requirements, **it eliminates the function from the overload set**. In this example, lines 11 and 12 call `multiply` with two `int` values and two `double` values, respectively. Of course, **all numeric types in C++ are copyable and support the * operator**, so the compiler can specialize `multiply` for each type. Interestingly, **you cannot call multiply with arguments of different types**, even if one argument's type can be implicitly converted to the other's. There's one type parameter, so both arguments must have identical types.

Using Incompatible Types with `multiply`

Err What if we pass to `multiply` arguments that **do not support copying or do not support the * operator** and there is no other function definition that matches the function call? The compiler will generate error messages. For example, the following code attempts to calculate the product of two strings:³⁰

³⁰. For your convenience, this code is provided in the source-code file as comments at the end of `main`.

[Click here to view code image](#)

```
std::string s1{ "hi" };
std::string s2{ "bye" };
auto result{multiply(s1, s2)}; // error: string does not support binary '*' operator
```

Class `string` supports copying but does not support the `*` operator. If we add these lines to Fig. 15.4's `main` and recompile, we get the following error messages from our preferred compilers—we added vertical spacing for readability. **Clang** produces:

[Click here to view code image](#)

```
fig15_04.cpp:7:17: error: invalid operands to binary
      return first * second;
                  ~~~~~ ^ ~~~~~
fig15_04.cpp:16:16: note: in instantiation of function
specialization
      auto result{multiply(s1, s2)}; // error: string
                                ^
1 error generated.
```

Visual C++ produces:

[Click here to view code image](#)

```
fig15_04.cpp
C:\Users\pauldeitel\examples\ch15\multiply\fig15_04.cpp: In
binary acceptable to the predefined operator
with
[
    T=std::string
]
C:\Users\pauldeitel\examples\ch15\multiply\fig15_04.cpp:16:31:
reference to function template instantiation being
with
[
    T=std::string
]
```

GNU g++ produces:

[Click here to view code image](#)

```
fig15_04.cpp: In instantiation of 'T multiply(T, T)
std::__cxx11::basic_string<char>]'':
fig15_04.cpp:16:31:     required from here
```

```
fig15_04.cpp:7:17: error: no match for 'operator*'  
`std::__cxx11::basic_string<char>' and 'std::__cxx1  
    7 |         return first * second;  
    |             ~~~~~^~~~~~
```

 The preceding errors are relatively small and straightforward, but this is not typical. More complex templates tend to result in lengthy lists of error messages. For example, passing the wrong kinds of iterators to a standard library algorithm that's not constrained with concepts can yield hundreds of lines of error messages. We tried to compile a program in which `main` contained only the following two simple statements:

[Click here to view code image](#)

```
std::list integers{10, 2, 33, 4, 7, 1, 80};  
std::sort(integers.begin(), integers.end());
```

We'd like to sort a `std::list`, which has **bidirectional iterators**. However, the `std::sort` algorithm requires **random-access iterators**. One popular compiler generated more than 1000 lines of error messages for the preceding code! Simply switching to the concept-constrained `std::ranges::sort` algorithm produces far fewer messages and indicates that **random-access iterators** are required.

20 15.4.2 Constrained Function Template with a C++20 Concepts **requires** Clause

  **C++20 concepts** enable you to specify **constraints**. Each constraint is a **compile-time predicate**—an expression that evaluates to true or false.³¹ The compiler uses constraints to check type requirements **before specializing templates**. For example, with predefined concepts, we can specify constraints that require a template's arguments to be numeric values of the same type. The C++ Core Guidelines recommend:

³¹. “Working Draft, Standard for Programming Language C++—13.5 Template Constraints.”

Accessed June 12, 2021. <https://eel.is/c++draft/temp.constr>.

- **specifying concepts for every template parameter**³² and

32. “T.10: Specify concepts for all template arguments.” Accessed June 9, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts>.

- **using the standard’s predefined concepts if possible.**³³

33. “T.11: Whenever possible use standard concepts.” Accessed June 9, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts>.

You can apply concepts to a function template parameter to explicitly state the requirements for the corresponding type arguments. If a type argument satisfies the requirements, the compiler specializes the template; otherwise, the template is ignored. When compilers do not find a match for a function call, **a benefit of concepts over previous techniques they either partially or wholly replace is that compilers typically produce (possibly far) fewer, clearer and more precise error messages.**

requires Clause

20 Figure 15.5 uses a C++20 **requires clause** (line 8) to constrain multiply’s template parameter T. The keyword **requires** is followed by a **constraint expression**, consisting of one or more compile-time bool expressions, which you can combine with the logical `&&`, `||` and `!` operators.

[Click here to view code image](#)

```
1 // fig15_05.cpp
2 // Constrained multiply function template t:
3 // only integers and floating-point values.
4 #include <concepts>
5 #include <iostream>
6
7 template<typename T>
8     requires std::integral<T> || std::floating_point<T>
9 T multiply(T first, T second) {
10     return first * second;
```

```
11     }
12
13     int main() {
14         std::cout << "Product of 5 and 3: " << m
15             << "\nProduct of 7.25 and 2.0: " << m
16
17         std::string s1{"hi"};
18         std::string s2{"bye"};
19         auto result{multiply(s1, s2)};
20     }
```

```
<          >
fig15_05.cpp: In function 'int main()':
fig15_05.cpp:19:31: error: use of function 'T mu
std::__cxx11::basic_string<char>]' with unsatisfi
19 |     auto result{multiply(s1, s2)};           ^
|
```

```
fig15_05.cpp:9:3: note: declared here
 9 | T multiply(T first, T second) {           ^
|   ~~~~~~
```

fig15_05.cpp:9:3: note: constraints not satisfied

```
fig15_05.cpp: In instantiation of 'T multiply(T,
std::__cxx11::basic_string<char>)':
fig15_05.cpp:19:31:   required from here
fig15_05.cpp:9:3:   required by the constraints
requires (integral<T>) || (floating_point<T>) T
```

```
fig15_05.cpp:8:30: note: no operand of the disju
 8 |     requires std::integral<T> || std::flo
|   ~~~~~~^~~~~~
```

Fig. 15.5 | Constrained multiply function template that allows only integers and floating-point values. These error messages are from the

g++ compiler.

Line 8 specifies that valid `multiply` type arguments must satisfy either of the following concepts, each of which is a **constraint on the type parameter T**:

- `std::integral<T>` indicates that **T can be any integer data type**, and
- `std::floating_point<T>` indicates that **T can be any floating-point data type**.

All integral and floating-point types support the arithmetic operators, so we know that values of these types all will work with line 10 in the template's body. If neither of the preceding constraints is satisfied, then the type arguments are incompatible with function template `multiply`, and the compiler will not instantiate the template. If no other functions match the call, the compiler generates error messages.

20 Concepts  You'll soon see that **some concepts specify many individual constraints**. The preceding concepts (from header `<concepts>`) are two of C++20's **74 predefined concepts**.³⁴ Section 15.4.3 lists the predefined-concept categories, the concepts in each and the headers that define them.

34. “Index of library concepts.” Accessed May 10, 2021.
<https://eel.is/c++draft/conceptindex>.

Disjunctions and Conjunctions

In line 8, the **logical OR (||) operator** forms a **disjunction**—either or both operands must be `true` for the compiler to specialize the template. If both are `false`, the compiler ignores the template as a potential match for a call to `multiply`. Function `multiply` defines only one type parameter, so **both operands must have the same type**. Thus, only one concept in line 8 can be `true` for each `multiply` call in this example.

You may form a **conjunction** with the **logical AND (&&) operator** to indicate that both operands must be `true` for the compiler to specialize the template. Disjunctions and conjunctions use **short-circuit evaluation** (Section 4.11.3). They also may contain the **logical NOT operator (!)**.

Calling `multiply` with Arguments That Satisfy Its Constraints

Lines 14 and 15 call `multiply` with two `int` values and two `double` values, respectively. When the compiler looks for function definitions that match these calls, it will encounter only the `multiply` function template in lines 7–11. It will check the arguments' types to determine whether they satisfy either of the concepts listed in line 8's **requires clause**. Both arguments in each call satisfy one of the **disjunction**'s requirements, so the compiler will specialize the template for type `int` in line 14 and type `double` in line 15. Again, **`multiply` has only one type parameter, so both arguments must be the same type**. Otherwise, the compiler will not know which type to use to specialize the template and will generate errors.

Calling `multiply` with Arguments That Do Not Satisfy Its Constraints

Err Line 19 calls `multiply` with two `string` arguments. Again, when the compiler looks for function definitions that match this call, it will encounter only the `multiply` function template in lines 7–11. Next, it will check the arguments' types to determine whether they satisfy either of the concepts listed in line 8's **requires clause**. **The `string` type does not satisfy either of the concepts**. Also, no other `multiply` functions can receive two `string` arguments, so the compiler generates error messages. [Figure 15.5](#) shows `g++`'s messages—we highlighted several messages in bold and added vertical spacing for readability. Note the last few lines of this output where the compiler indicated that "**no operand of the disjunction is satisfied**" and pointed to the **requires clause** in line 8.

20 15.4.3 C++20 Predefined Concepts

Concepts The C++ standard's “Index of library concepts”³⁵ alphabetically lists the concepts defined in the standard. In the following table, we've listed all the standard concepts by header—`<concepts>`, `<iterator>`, `<ranges>`, `<compare>` and `<random>`. Also, we've divided the `<concepts>` header's 31 concepts into the subcategories **core language concepts**, **comparison concepts**, **object concepts** and **callable concepts**. Many concepts have self-explanatory names. For details on each,

see the corresponding header's page at cppreference.com.

35. "Working Draft, Standard for Programming Language C++—Index of library concepts." Accessed May 15, 2021. <https://eel.is/c++draft/conceptindex>.

74 predefined C++20 concepts

<concepts> header core language concepts

assignable_from	default_initializable	same_as
common_reference_with	derived_from	signed_integral
common_with	destructible	swappable
constructible_from	floating_point	swappable_with
convertible_to	integral	unsigned_integral
copy_constructible	move_constructible	

<concepts> header comparison concepts

equality_comparable	totally_ordered	totally_ordered_with
equality_comparable_with		

<concepts> header object concepts

copyable	regular	semiregular
movable		

<concepts> header callable concepts

equivalence_relation	predicate	relation
invocable	regular_invocable	strict_weak_order

<iterator> header concepts

bidirectional_iterator	indirectly_copyable_storable	input_or_output_iterator
contiguous_iterator	indirectly_movable	mergeable
forward_iterator	indirectly_movable_storable	output_iterator
incrementable	indirectly_readable	permutable
indirect_binary_predicate	indirectly_regular_unary_invocable	random_access_iterator
indirect_equivalence_relation	indirectly_swappable	sentinel_for
indirect_strict_weak_order	indirectly_unary_invocable	sized_sentinel_for
indirect Unary_predicate	indirectly_writable	sortable
indirectly_comparable	input_iterator	weakly_incrementable
indirectly_copyable		

<ranges> header concepts

bidirectional_range	forward_range	random_access_range
borrowed_range	input_range	sized_range
common_range	output_range	view_range
contiguous_range	range	viewable_range

<compare> header concepts

three_way_comparable	three_way_comparable_with
----------------------	---------------------------

<random> header concept

uniform_random_bit_generator

15.5 Type Traits

11 C++11 introduced the `<type_traits>` header³⁶ for

36. “Standard library header `<type_traits>`.” Accessed June 14, 2021.
https://en.cppreference.com/w/cpp/header/type_traits.

- testing at compile time whether types have various traits, and
- generating template code based on those traits.

For example, you could check whether a type is

- a **fundamental type** like `int` (using the type trait `std::is_fundamental`) vs.
- a **class type** (using the type trait `std::is_class`)

20 and use different template code to handle each case. Each subsequent C++ version has added more type traits—a few have been deprecated or removed (which we do not include in our table at the end of this section). Most recently, **C++20 added 10 type traits**.

Using Type Traits Before C++20

Before concepts, you’d use type traits in unconstrained template definitions to check whether type arguments satisfied a template’s requirements. As with concepts, these checks were performed at compile-time **but during template specialization**, often leading to large numbers of difficult-to-decipher error messages. On the other hand, **the compiler tests concepts before performing template specialization**, typically resulting in fewer, clearer and more focused error messages than when you use type traits in unconstrained templates.

20 C++20 Predefined Concepts Often Use Type Traits

Interestingly, C++20 concepts are often implemented in terms of type traits, for example:

- the **concept** `std::integral` is implemented in terms of the **type trait** `std::is_integral`,
- the **concept** `std::floating_point` is implemented in terms of the

type trait `std::is_floating_point` and

- the **concept** `std::destructible` is implemented in terms of the **type trait** `std::is_nothrow_destructible`.

Demonstrating Type Traits

Figure 15.6 shows **type traits** that correspond to the concepts in Fig. 15.5.

[Click here to view code image](#)

```
1 // fig15_06.cpp
2 // Using type traits to test whether types are
3 // integral types, floating-point types or
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <string>
7 #include <type_traits>
8
9 int main() {
10     std::cout << fmt::format("{}\n{}{}\n{}{}")
11                 "CHECK WITH TYPE TRAITS WHETHER TYPES"
12                 "std::is_integral<int>::value: ", std::
13                 "std::is_integral_v<int>: ", std::is_
14                 "std::is_integral_v<long>: ", std::is_
15                 "std::is_integral_v<float>: ", std::is_
16                 "std::is_integral_v<std::string>: ",
17                 std::is_integral_v<std::string>);
18
19     std::cout << fmt::format("{}\n{}{}\n{}{}")
20                 "CHECK WITH TYPE TRAITS WHETHER TYPES"
21                 "std::is_floating_point<float>::value"
22                 std::is_floating_point<float>::value,
23                 "std::is_floating_point_v<float>: ",
24                 std::is_floating_point_v<float>,
25                 "std::is_floating_point_v<double>: ",
26                 std::is_floating_point_v<double>,
27                 "std::is_floating_point_v<int>: ",
```

```
28     std::is_floating_point_v<int>,
29     "std::is_floating_point_v<std::string>
30     std::is_floating_point_v<std::string>
31
32     std::cout << fmt::format("{}\n{}{}\n{}{}")
33         "CHECK WHETHER TYPES ARE INTEGRAL\n"
34         "std::is_arithmetic<int>::value: ", s
35         "std::is_arithmetic_v<int>: ", std::is
36         "std::is_arithmetic_v<double>: ", std::is
37         "std::is_arithmetic_v<std::string>: "
38             std::is_arithmetic_v<std::string>);
39 }
```

< >

```
CHECK WHETHER TYPES ARE INTEGRAL
std::is_integral<int>::value: true
std::is_integral_v<int>: true
std::is_integral_v<long>: true
std::is_integral_v<float>: false
std::is_integral_v<std::string>: false
```

```
CHECK WHETHER TYPES ARE FLOATING POINT
std::is_floating_point<float>::value: true
std::is_floating_point_v<float>: true
std::is_floating_point_v<double>: true
std::is_floating_point_v<int>: false
std::is_floating_point_v<std::string>: false
```

```
CHECK WHETHER TYPES CAN BE USED IN ARITHMETIC
std::is_arithmetic<int>::value: true
std::is_arithmetic_v<int>: true
std::is_arithmetic_v<double>: true
std::is_arithmetic_v<std::string>: false
```

Fig. 15.6 | Using type traits to test whether types are integral types, floating-point types or arithmetic types. (Part 2 of 2.)

Each type-trait class has a `static bool` member named `value`. An expression like

[Click here to view code image](#)

```
std::is_integral<int>::value
```

17.14 (line 12) returns `true` at compile time if the type in angle brackets (`int`) is an integral type. Otherwise, it returns `false`. Since the notation `::value` is used with every type-trait class to access its `true` or `false` value, **C++17 added convenient shorthands for using type trait values.** These are defined as **variable templates** (a C++14 feature) with **names ending in `_v`**.³⁷ Just like function templates specify groups of related functions and class templates specify groups of related classes, variable templates specify groups of related variables. You specialize a variable template to use it. The **variable template** corresponding to

37. “T.142: Use template variables to simplify notation.” Accessed June 18, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-var>.

[Click here to view code image](#)

```
std::is_integral<int>::value
```

is defined in the C++ standard as³⁸

38. “Working Draft, Standard for Programming Language C++—20.15.3 Header `<type_traits>` synopsis [meta.type.synop],” Accessed June 18, 2021. <https://eel.is/c++draft/meta#type.synop>.

[Click here to view code image](#)

```
template<class T>
inline constexpr bool is_integral_v = is_integral·
< >
```

So, the expression `std::is_integral<int>::value` is equivalent to

```
std::is_integral_v<int>
```

which, depending on whether the type is `int`, simply becomes the value `true` or `false` in the generated code.

Lines 12, 22 and 34 tests several type traits and display their value

members to show the results. The program's other tests are performed using the more convenient **_v variable templates**:

- Lines 13, 14, 15 and 17 use the variable template `std::is_integral` to check whether various types are **integer types**.
- Lines 24, 26, 28 and 30 use the variable template `std::is_floating_point` to check whether various types are **floating-point types**.
- Lines 35, 36 and 38 use the variable template `std::is_arithmetic_v` to check whether various types are **arithmetic types** that could be used in arithmetic expressions.

The following table lists the `<type_traits>` header's type traits and supporting functions by category. The new features added in C++14 (two new items), 17 (14 new items) and 20 (10 new items) are indicated with superscript version numbers. For details on each, see:

[Click here to view code image](#)

https://en.cppreference.com/w/cpp/header/type_traits



Type traits by category

Helper Classes

<code>bool_constant</code> ²⁰	<code>false_type</code>	<code>true_type</code>
<code>integral_constant</code>		

Primary type categories

<code>is_void</code>	<code>is_enum</code>	<code>is_lvalue_reference</code>
<code>is_null_pointer</code> ¹⁴	<code>is_union</code>	<code>is_rvalue_reference</code>
<code>is_integral</code>	<code>is_class</code>	<code>is_member_object_pointer</code>
<code>is_floating_point</code>	<code>is_function</code>	<code>is_member_function_pointer</code>
<code>is_array</code>	<code>is_pointer</code>	

Composite type categories

<code>is_fundamental</code>	<code>is_object</code>	<code>is_reference</code>
<code>is_arithmetic</code>	<code>is_compound</code>	<code>is_member_pointer</code>
<code>is_scalar</code>		

Type properties

<code>has_unique_object_representations</code> ¹⁷	<code>is_standard_layout</code>	<code>is_aggregate</code> ¹⁷
	<code>is_empty</code>	<code>is_signed</code>
<code>is_const</code>	<code>is_polymorphic</code>	<code>is_unsigned</code>
<code>is_volatile</code>	<code>is_abstract</code>	<code>is_bounded_array</code> ²⁰
<code>is_trivial</code>	<code>is_final</code> ¹⁴	<code>is_unbounded_array</code> ²⁰
<code>is_trivially_copyable</code>		

Supported operations

<code>is_constructible</code>	<code>is_move_constructible</code>	<code>is_trivially_move_assignable</code>
<code>is_trivially_constructible</code>	<code>is_trivially_move_constructible</code>	<code>is_nothrow_move_assignable</code>
<code>is_nothrow_constructible</code>	<code>is_nothrow_move_constructible</code>	<code>is_destructible</code>
<code>is_default_constructible</code>	<code>is_nothrow_move_constructible</code>	<code>is_trivially_destructible</code>
<code>is_trivially_default_constructible</code>	<code>is_assignable</code>	<code>is_nothrow_destructible</code>
	<code>is_trivially_assignable</code>	<code>has_virtual_destructor</code>
<code>is_nothrow_default_constructible</code>	<code>is_nothrow_assignable</code>	<code>is_swappable_with</code> ¹⁷
	<code>is_copy_assignable</code>	<code>is_swappable</code> ¹⁷
<code>is_copy_constructible</code>	<code>is_trivially_copy_assignable</code>	<code>is_nothrow_swappable_with</code> ¹⁷
<code>is_trivially_copy_constructible</code>	<code>is_nothrow_copy_assignable</code>	<code>is_nothrow_swappable</code> ¹⁷
<code>is_nothrow_copy_constructible</code>	<code>is_move_assignable</code>	

Property queries		
alignment_of	rank	extent
Type relationships		
is_same	is_layout_compatible ²⁰	is_invocable_r ¹⁷
is_base_of	is_pointer_interconvert-	is_nothrow_invocable ¹⁷
is_convertible	ible_base_of ²⁰	is_nothrow_invocable_r ¹⁷
is_nothrow_convertible ²⁰	is_invocable ¹⁷	
Const-volatility specifiers		
remove_cv	remove_volatile	add_const
remove_const	add_cv	add_volatile
References		
remove_reference	add_lvalue_reference	add_rvalue_reference
Pointers		
remove_pointer	add_pointer	
Sign modifiers		
make_signed	make_unsigned	
Arrays		
remove_extent	remove_all_extents	
Miscellaneous transformations		
aligned_storage	enable_if	underlying_type
aligned_union	conditional	invoke_result ¹⁷
decay	common_type	void_t ¹⁷
remove_cvref ²⁰		
Operations on traits		
conjunction ¹⁷	disjunction ¹⁷	negation ¹⁷
Functions		
Member relationships		
is_pointer_interconvertible_with_class ²⁰		is_corresponding_member ²⁰
Constant evaluation context		
is_constant_evaluated ²⁰		

20 15.6 C++20 Concepts: A Deeper Look

Now that we've introduced how to constrain a template parameter via the **requires clause** and **predefined concepts**, we'll create a **custom concept** that aggregates two predefined ones, then present additional concepts features.

Concepts © 15.6.1 Creating a Custom Concept

C++20's **predefined concepts often aggregate multiple constraints, sometimes including those from other predefined concepts**, effectively creating what some developers refer to as a **type category**.³⁹ You also can use templates to define your own custom concepts. Let's create a **custom concept** that aggregates the predefined concepts `std::integral` and `std::floating_point`, we used in Fig. 15.5 to constrain the `multiply` function template:

³⁹. Inbal Levi. "Exploration of C++20 Metaprogramming." September 29, 2020. Accessed June 14, 2021. <https://www.youtube.com/watch?v=XgrjybKaIV8>.

[Click here to view code image](#)

```
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

A concept begins with a template header followed by:

- 20 the keyword **concept**,
- the **concept name** (`Numeric`), which uses CamelCase naming by convention,
- an **equal sign (=)** and
 - the **constraint expression** (`std::integral<T> || std::floating_point<T>`).

The **constraint expression** is a compile-time logical expression that determines whether a template argument satisfies a particular set of **requirements**—in this case, whether it's an integer or floating-point number. These commonly include **predefined concepts**, **type traits** and, as you'll see

shortly, **requires expressions** that enable you to specify other requirements. The `Numeric` concept's template header has one type parameter, representing the type to test.

Concepts with multiple template parameters also can specify relationships between types.⁴⁰ For example, if a template has two type parameters, you can **use the predefined concept `std::same_as` to ensure two type arguments have the same type**, as in:

40. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying template interfaces,” January 31, 2017. Accessed May 8, 2021. <http://wg21.link/p0557r0>.

[Click here to view code image](#)

```
template<typename T, typename U>
    requires std::same_as<T, U>
// rest of template definition
```

In [Section 15.12](#), we introduce **variadic templates** that can have any number of (type or non-type) template arguments. There, we'll use `std::same_as` in a variadic function template that requires all its arguments to have the same type.

15.6.2 Using a Concept

Concepts Once you define a concept, you can use it to **constrain a template parameter** in one of four ways—we show the first three here and the fourth in [Section 15.6.3](#).

[requires Clause Following the template Header](#)

Any concept can be placed in a `requires` clause **following the template header**, as we did in [Fig. 15.5](#). We can use our **custom concept `Numeric`** in a `requires` clause as follows:

[Click here to view code image](#)

```
template<typename T>
    requires Numeric<T>
T multiply(T first, T second) {
    return first * second;
}
```

requires Clause Following a Function Template's Signature

You also can place the **requires** clause after a function template's signature and before the function template's body, as in:

[Click here to view code image](#)

```
template<typename T>
T multiply(T first, T second) requires Numeric<T>
    return first * second;
}
```



Such a **trailing requires clause** must be used in two scenarios:⁴¹

41. "What is the difference between the three ways of applying constraints to a template?" Accessed June 12, 2021. <https://stackoverflow.com/a/61875483>. [Note: The original stackoverflow.com question mentioned three ways of applying constraints to a template, but there are four and each is mentioned in the cited answer.]

- A member function defined in a class template's body does not have a template header, so you must use a trailing requires clause.
- To use a function template's parameter names in a constraint, you must use a trailing requires clause so the parameter names are in scope.

Concept as a Type in the **template** Header

 When you have a single concept-constrained type parameter, the C++ Core Guidelines recommend using the concept name in place of typename in the template header.⁴² Doing so simplifies the template definition by **eliminating the requires clause**:

42. "T.13: Prefer the shorthand notation for simple, single-type argument concepts." Accessed June 10, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-shorthand>.

[Click here to view code image](#)

```
template<Numeric Number>
Number multiply(Number first, Number second) {
    return first * second;
}
```

 In this case, we also replaced the generic type-parameter name **T** with the more descriptive type-parameter name **Number**. Now, each parameter clearly indicates that it's required to satisfy the requirements of concept **Numeric**. The function template still has only one type parameter (**Number**), so the functions arguments must have the same type; otherwise, a compilation error occurs.

15.6.3 Using Concepts in Abbreviated Function Templates

 SE  Figure 15.3 introduced **abbreviated function templates** with the parameters declared **auto** so the compiler can infer the function's parameter types. **Anywhere you can use auto in your code, you can precede it with a concept name to constrain the allowed types.**⁴³ This includes

43. “Placeholder type specifiers.” Accessed June 14, 2021. <https://en.cppreference.com/w/cpp/language/auto>.

- abbreviated function template return types and parameter lists,
- **auto** local-variable definitions that infer a variable's type from an initializer, and
- generic lambda expressions.

Figure 15.7 reimplements `multiply` as an **abbreviated function template**. In this case, we used **constrained auto** for each parameter, using our **Numeric concept** to restrict the types we can pass as arguments. We also **used auto as the return type**, so the compiler will infer it from the type of the expression in line 12.

[Click here to view code image](#)

```
1 // fig15_07.cpp
2 // Constrained multiply abbreviated function template
3 #include <concepts>
4 #include <iostream>
5
6 // Numeric concept aggregates std::integral
```

```

7  template<typename T>
8  concept Numeric = std::integral<T> || std::
9
10 // abbreviated function template with const
11 auto multiply(Numeric auto first, Numeric a
12     return first * second;
13 }
14
15 int main() {
16     std::cout << "Product of 5 and 3: " << m
17         << "\nProduct of 7.25 and 2.0: " << m
18         << "\nProduct of 5 and 7.25: " << mul
19 }
```



Product of 5 and 3: 15
 Product of 7.25 and 2.0: 14.5
 Product of 5 and 7.25: 36.25

Fig. 15.7 | Constrained multiply abbreviated function template.

Err The key difference between this **abbreviated function template** and the constrained multiply function templates in [Section 15.6.2](#) is that the compiler treats each **auto** in line 11 as a **separate template type parameter**. Thus, **first** and **second** **can have different data types**, as in line 18, which passes an **int** and a **double**. Our previous **concept-constrained version** of multiply would generate compilation errors for arguments of different types. Lines 11–13 are actually equivalent to a template with **two type parameters**:

[Click here to view code image](#)

```

template<Numeric Number1, Numeric Number2>
auto multiply(Number1 first, Number2 second) {
    return first * second;
}
```

SE  If you require the same type for two or more parameters,

- use a regular function template rather than an abbreviated function template, and
- use the same type parameter name for every function parameter that must have the same type.

15.6.4 Concept-Based Overloading

Concepts  **SE**  Function templates and overloading are intimately related. **When overloaded functions perform identical operations on different types, they can be expressed more compactly and conveniently using function templates.** You can then write function calls with different argument types and let the compiler generate separate **specializations** to handle each function call appropriately. The specializations all have the same name, so the compiler uses **overload resolution** to invoke the proper function.

Matching Process for Overloaded Functions

When determining what function to call, the compiler looks at functions and function templates to locate an existing function or generate a function-template specialization that matches the call:

- **If there are no matches**, the compiler issues an **error message**.
- **If there are multiple matches** for the function call, the compiler attempts to determine the **best match**.
-  If there's **more than one best match**, the call is **ambiguous**, and the compiler issues an **error message**.⁴⁴

⁴⁴. The compiler's process for resolving function calls is complex. The complete details are discussed in "Working Draft, Standard for Programming Language C++—12.2 Overload Resolution." Accessed June 12, 2021. <https://eel.is/c++draft/over.match>.

Overloading Function Templates

You also may **overload function templates**. For example, you can provide other function templates with different signatures. A function template also can be overloaded by providing non-template functions with the same function name but different parameters.

Overloading Function Templates Using Concepts

20 With C++20, you can use concepts in function templates to select overloads based on type requirements.⁴⁵ Consider the standard functions `std::distance` and `std::advance` from the `<iterator>` header:

45. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—6 Concept Overloading,” January 31, 2017. Accessed June 14, 2021. <http://wg21.link/p0557r0>.

- `std::distance` calculates the number of container elements between two iterators.⁴⁶

46. “`std::distance`.” Accessed June 14, 2021.
<https://en.cppreference.com/w/cpp/iterator/distance>.

- `std::advance` advances an iterator n positions from its current location in a container.⁴⁷

47. “`std::advance`.” Accessed June 14, 2021.
<https://en.cppreference.com/w/cpp/iterator/advance>.

Each of these can be implemented as $O(n)$ operations:

- `std::distance` can use `++` to iterate from a **begin iterator** up to but not including an **end iterator** and count the number of increments (n).
- `std::advance` can loop n times, incrementing the iterator once per iteration of the loop.

Some algorithms can be optimized for specific iterator types—both `std::distance` and `std::advance` can be implemented as $O(1)$ operations for **random-access iterators**:

- `std::distance` can use `operator-` to subtract a **begin iterator** from an **end iterator** to calculate the number of items between the iterators in one operation.
- `std::advance` can use `operator+` to add an integer n to an iterator, advancing the iterator n positions in one operation.

Figure 15.8 implements overloaded `customDistance` **abbreviated function templates**. Each requires two iterator arguments and calculates the number of elements between them. We use **concept-based overloading** (also called **concept overloading**) to enable the compiler to choose the correct overload. **The compiler chooses between the overloads based on the**

concepts we use to constrain each template's parameters. To distinguish between the function templates, we use the predefined `<iterator>` header concepts `std::random_access_iterator` and `std::input_iterator`:

- The **$O(n)$ customDistance function** in lines 11–22 requires two arguments that satisfy the `std::input_iterator` concept.
- The **$O(1)$ customDistance function** in lines 26–30 requires two arguments that satisfy the `std::random_access_iterator` concept.

[Click here to view code image](#)

```
1 // fig15_08.cpp
2 // Using concepts to select overloads.
3 #include <array>
4 #include <iostream>
5 #include <iterator>
6 #include <list>
7
8 // calculate the distance (number of items)
9 // using input iterators; requires incrementation
10 // so this is an  $O(n)$  operation
11 auto customDistance(std::input_iterator auto begin,
12                     std::input_iterator auto end) {
13     std::cout << "Called customDistance with ";
14     std::ptrdiff_t count{0};
15
16     // increment from begin to end and count
17     for (auto& iter{begin}; iter != end; ++iter)
18         ++count;
19 }
20
21     return count;
22 }
23
24 // calculate the distance (number of items)
```

```

25 // using random-access iterators and an O(1)
26 auto customDistance(std::random_access_iterator
27     std::random_access_iterator auto end) {
28     std::cout << "Called customDistance with "
29     return end - begin;
30 }
31
32 int main() {
33     std::array ints1{1, 2, 3, 4, 5}; // has
34     std::list ints2{1, 2, 3}; // has bidirec
35
36     auto result1{customDistance(ints1.begin(
37         std::cout << "ints1 number of elements: "
38         auto result2{customDistance(ints2.begin(
39             std::cout << "ints2 number of elements: "
40     })

```

```

Called customDistance with random-access iterator
ints1 number of elements: 5
Called customDistance with bidirectional iterator
ints2 number of elements: 3

```

Fig. 15.8 | Using concepts to select overloads.

SE A Lines 33–34 define an array and a list. Line 36 calls `customDistance` for the array `ints1`, which has **random-access iterators**. Of course, our function that requires input iterators also could receive random-access iterators. However, **when multiple function templates satisfy a given function call, as in this example, the compiler calls the most constrained version**,^{48,49} so line 36 calls the **$O(1)$ version of `customDistance`** in lines 26–30. For the call in line 38, a list's **bidirectional iterators** do not satisfy the constraints of the function template in lines 26–30. So, that version is eliminated from consideration, and the slower $O(n)$ `customDistance` in lines 11–22 is called.

48. Hendrik Niemeyer, “An Introduction to C++20’s Concepts,” July 25, 2020. Accessed June 1, 2021. https://www.youtube.com/watch?v=N_kPd2OK1L8.
49. “Working Draft, Standard for Programming Language C++—13.5 Template Constraints.” Accessed June 12, 2021. <https://eel.is/c++draft/temp.constr>.

15.6.5 **requires** Expressions

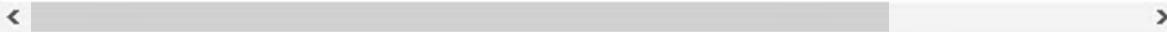
Concepts  The **C++ Core Guidelines** recommend using **standard concepts**.⁵⁰ For custom requirements that cannot be expressed via the standard ones, you can use a **requires expression**, which has two forms:

50. “T.11: Whenever possible use standard concepts.” Accessed June 7, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts>.

```
requires {  
    requirements-definitions  
}
```

or

```
requires (parameter-list) {  
    requirements-definitions that optionally use t.  
}
```



The *parameter-list* looks like a function’s parameter list but **cannot have default arguments**. The compiler uses a **requires** expression’s parameters only to check whether types satisfy the requirements defined in the expression’s braces. The braces may contain any combination of the four requirement types described below—**simple**, **type**, **compound** and **nested**. Each requirement ends with a semicolon (;).

Simple Requirements

20 A **simple requirement** checks whether an expression is valid. Consider the definition of the <ranges> header’s **range concept**, which checks whether an object’s type represents a **C++20 range** with a **begin iterator** and an **end sentinel**:⁵¹

51. “Working Draft, Standard for Programming Language C++—24.4.2 Ranges.” Accessed June 10,

2021. <https://eel.is/c++draft/ranges#range.range>.

[Click here to view code image](#)

```
1  template<class T>
2      concept range =
3          requires(T& t) {
4              std::ranges::begin(t);
5              std::ranges::end(t);
6      };
```

The requirements in a **requires expression** can reference the expression's parameters and any template parameters from the template header. This **range concept** defines a parameter *t* of type *T&*. Lines 4 and 5 define **simple requirements** indicating that *t* is a *range* only if we can get *t*'s **begin iterator** and **end sentinel**, respectively, by passing *t* to the functions **std::ranges::begin** and **std::ranges::end**. If either of these **simple requirements** evaluates to *false* at compile time, then *t* is not a *range*.

Simple requirements can specify operator expressions. The following concept specifies operations that would be expected of any integer or floating-point arithmetic type:

[Click here to view code image](#)

```
template<class T>
concept ArithmeticType =
    requires(T a, T b) {
        a + b;
        a - b;
        a * b;
        a / b;
        a += b;
        a -= b;
        a *= b;
        a /= b;
    };
```

We did not include the modulus (%) operator because it requires integer operands. Of course, built-in arithmetic types support more operations, such as implicit conversions between types, so a “real” **ArithmeticType concept** would be more elaborate than this. In addition, C++ already has the type trait **is_arithmetic** to test for arithmetic types.

Type Requirements

A **type requirement** starts with typename followed by a type and determines whether the specified type is valid. For example, if your code requires that a template type parameter be used to specialize a vector, you might define a concept with a type requirement like:

[Click here to view code image](#)

```
template<typename T>
concept WorksWithVector = requires {
    typename std::vector<T>;
};
```

A type T would satisfy WorksWithVector only if **std::vector<T>** is a valid type. If the compiler cannot specialize vector for type T, this requirement would evaluate to false.

Compound Requirements

A **compound requirement** allows you to specify an expression that also has requirements on its return value. Such requirements have the form

{ *expression* } -> *return-type-requirement*

For example, the standard library <iterator> header’s **incrementable concept** contains the following requires expression:⁵²

52. “Working Draft, Standard for Programming Language C++—23.3.4.5 Concept incrementable.” Accessed June 10, 2021. <https://eel.is/c++draft/iterator.concepts#iterator.concept.inc>.

```
requires(I i) {
    { i++ } -> same_as<I>;
}
```

This requirement indicates that an **incrementable iterator** must support

the postincrement operator (++). **The `->` notation indicates a requirement on the `i++` expression's result.** In this case, the result must have the same type (`I`) as the object specified in the `requires` expression's parameter list. The right brace in a compound requirement optionally may be followed by `noexcept` to indicate that the expression is not allowed to throw exceptions.

An `incrementable` object also must support the `concept weakly_incrementable`, which among its other requirements contains the following **compound requirement**:

```
{ ++i } -> same_as<I>;
```

So, `incrementable` objects also must support pre-incrementing and the expression's result must have the same type as the type parameter `I`.

Nested Requirement

A **nested requirement** is a `requires` clause nested in a `requires expression`. You'd use these to apply existing concepts and type traits to the `requires` expression's parameters.

requires requires—Ad-Hoc Constraints

A `requires expression placed directly in a requires clause` is an **ad-hoc constraint**. In the following `printRange` function template, we copied the `std::range` concept's `requires` expression and placed it directly in a `requires` clause:

[Click here to view code image](#)

```
template<typename T>
    requires requires(T& t) {
        std::ranges::begin(t);
        std::ranges::end(t);
    }
void printRange(const T& range) {
    for (const auto& item : range) {
        std::cout << item << " ";
    }
}
```

The notation `requires` `requires` is correct:

- The first `requires` introduces the **requires clause** and
- the second `requires` introduces the **requires expression**.

SE A The benefit of an ad-hoc constraint is that if you need it only once, you can define it where it's used. **Named concepts are preferred for reuse.**

20 15.6.6 C++20 Exposition-Only Concepts

Concepts C Throughout the C++ standard document, the phrases “**for the sake of exposition**” or “**exposition only**” appear over 400 times. These are displayed in *italics* and indicate **items used only for discussion purposes**⁵³—often to show how something can be implemented. For example, the standard uses the following **exposition-only has-arrow concept** in the ranges library to describe an iterator type that supports the operator `->:`⁵⁴

53. “Exposition-only in the C++ standard?” Answered December 28, 2015. Accessed May 10, 2021. <https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard>.

54. “24.5.2 Helper concepts [range.utility.helpers].” Accessed June 25, 2021. <https://eel.is/c++draft/range.utility>.

[Click here to view code image](#)

```
template<class I>
concept has-arrow =
    input_iterator<I> && (is_pointer_v<I> || requires(I i) { i.operator->(); });
```

Similarly, the standard uses the following **exposition-only decrementable concept** in the ranges library to describe iterator types that support the `--` operator:⁵⁵

55. “24.6.4.2 Class template iota_view [range.iota.view].” Accessed June 25, 2021. <https://eel.is/c++draft/range.factories#range.iota.view-2>.

[Click here to view code image](#)

```
template<class I>
```

```
concept decrementable =
    incrementable<I> && requires(I i) {
        { --i } -> same_as<I&>;
        { i-- } -> same_as<I>;
    };
```

You may wonder why **decrementable** requires *incrementable*. Recall that all iterators support `++`, so a *decrementable* iterator also must be *incrementable*.

20 In addition to the 74 predefined C++20 concepts, the standard uses 31 exposition-only concepts shown in the following table. You can find each through the standard’s “Index of library concepts.”⁵⁶

56. “Working Draft, Standard for Programming Language C++—Index of library concepts.” Accessed May 15, 2021. <https://eel.is/c++draft/conceptindex>.

31 exposition-only concepts

C++20 concepts library

<i>boolean-testable</i>	<i>same-as-impl</i>
<i>boolean-testable-impl</i>	<i>weakly-equality-comparable-with</i>

Iterators library

<i>can-reference</i>	<i>cpp17-input-iterator</i>	<i>dereferenceable</i>
<i>cpp17-bidirectional-iterator</i>	<i>cpp17-iterator</i>	<i>indirectly-readable-impl</i>
<i>cpp17-forward-iterator</i>	<i>cpp17-random-access-iterator</i>	<i>simple-view</i>

C++20 ranges library

<i>advanceable</i>	<i>has-tuple-element</i>	<i>pair-like-convertible-from</i>
<i>convertible-to-non-slicing</i>	<i>iterator-sentinel-pair</i>	<i>stream-extractable</i>
<i>decrementable</i>	<i>not-same-as</i>	<i>tiny-range</i>
<i>has-arrow</i>	<i>pair-like</i>	

Comparisons in the language support library

<i>compares-as</i>	<i>partially-ordered-with</i>
--------------------	-------------------------------

Memory library

<i>no-throw-forward-iterator</i>	<i>no-throw-input-iterator</i>	<i>no-throw-sentinel</i>
<i>no-throw-forward-range</i>	<i>no-throw-input-range</i>	

15.6.7 Techniques Before C++20 Concepts: SFINAE and Tag Dispatch

With each new version of C++, metaprogramming gets more powerful and convenient. There's a history of several technologies that led to C++20 concepts, including **SFINAE**, **tag dispatch** and **constexpr if**. (We show an example with `constexpr if` in [Section 15.13.3](#).) For an overview of the progression through these technologies and how C++20 concepts simplify your code, see the blog post, “Notes on C++ SFINAE, Modern C++ and C++20 Concepts.”⁵⁷

⁵⁷. Bartek Filipek, “Notes on C++ SFINAE, Modern C++ and C++20 Concepts,” April 20, 2020. Accessed June 27, 2021. <https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html>.

SFINAE—Substitution Failure Is Not an Error

Earlier we mentioned that when you call a function, the compiler locates all the functions that might satisfy the function call—known as the **overload set**. From these overloads, the compiler chooses the best match. This process often includes **specializing function templates** based on a function call's argument types.

Before concepts, template metaprogramming techniques involving `std::enable_if` were commonly used with **type traits** to check if a function-template argument satisfied a template's requirements. If not, the compiler would generate an invalid function-template specialization. It would then ignore that specialization, removing it from the overload set. **SFINAE (substitution failure is not an error)**^{58,59,60} describes how the compiler discards invalid template specialization code as it tries to determine the correct function to call. This technique prevents the compiler from immediately generating (potentially) lengthy lists of error messages when first specializing a template. The compiler generates error messages only if it cannot find a match for the function call in the overload set.

58. “Substitution failure is not an error.” Accessed February 4, 2021. <https://w.wiki/yFe>.
59. Bartłomiej Filipek. “Notes on C++ SFINAE, Modern C++ and C++20 Concepts.” Accessed February 4, 2021. <https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html>.
60. David Vandevoorde and Nicolai M. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

Tag Dispatch

20 You can tell the compiler the version of an overloaded function to call based not only on template type parameters but also on properties of those types using the **tag-dispatch**⁶¹ technique. Bjarne Stroustrup—in his paper “Concepts: The Future of Generic Programming”—refers to properties of types as “concepts” and calls the tag-dispatching technique **concept overloading** or **concept-based overloading**.⁶² He then discusses how C++20 concept-based overloading (Section 15.6.4) can replace tag dispatch.

61. “Generic Programming—Tag Dispatching.” Accessed January 31, 2021. https://www.boost.org/community/generic_programming.html#tag_dispatching
62. Bjarne Stroustrup. “Concepts: The Future of Generic Programming (Section 6).” Accessed January 31, 2021. https://www.stroustrup.com/good_concepts.pdf.

20 15.7 Testing C++20 Concepts with `static_assert`

Concepts C 11 Concepts produce compile-time `bool` values, which you can test at compile-time with a C++11 `static_assert` declaration.⁶³ `static_assert` was developed to add compile-time assertion support for reporting incorrect usage of template libraries.⁶⁴ Figure 15.9 tests our custom `Numeric` concept from Section 15.6.1.

63. “T.150: Check that a class matches a concept using `static_assert`.” Accessed June 18, 2021.

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-check-class>.

64. Robert Klarer, Dr. John Maddock, Brian Dawes and Howard Hinnant, “Proposal to Add Static Assertions to the Core Language (Revision 3),” October 20, 2004. Accessed July 1, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html>.

[Click here to view code image](#)

```
1 // fig15_09.cpp
2 // Testing custom concepts with static_assert
3 #include <iostream>
4 #include <string>
5
6 template<typename T>
7 concept Numeric = std::integral<T> || std::
8
9 int main() {
10     static_assert(Numeric<int>); // OK: int
11     static_assert(Numeric<double>); // OK: double
12     static_assert(Numeric<std::string>); // Error: string
13 }
```

< >

```
fig15_09.cpp:12:4: error: static_assert failed
```

```
static_assert(Numeric<std::string>); // error
^
~~~~~  
  
fig15_09.cpp:12:18: note: because does not satisfy
    static_assert(Numeric<std::string>); // error
           ^  
  
fig15_09.cpp:7:24: note: because
concept Numeric = std::integral<T> || std::float
           ^  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../...
concepts:102:24: note: because evaluated to false
    concept integral = is_integral_v<_Tp>;
           ^  
  
fig15_09.cpp:7:44: note: and
concept Numeric = std::integral<T> || std::float
           ^
concepts:111:30: note: because evaluated to false
    concept floating_point = is_floating_point_v<
           ^  
  
1 error generated.
```

Fig. 15.9 | Testing custom concepts with `static_assert`. (Part 2 of 2.)

Err When the `static_assert` argument is false, the compiler outputs an error message. If `static_assert` argument is true, the compiler does not output any messages—it simply continues compiling the code. The expression in line 10

```
static_assert(Numeric<int>);
```

specializes the **Numeric concept** with type `int`. If `int` satisfies `Numeric`'s requirements, which it does, `Numeric<int>` evaluates to `true`, so the compiler continues compiling the code. Similarly, line 11's `Numeric<double>` also evaluates to `true`.

However, `Numeric<std::string>` in line 12 evaluates to `false` because a `string` is not `Numeric`. The output window shows the error messages produced by the **Clang C++** compiler. We highlighted the key messages in bold and added blank lines for readability. For the **Numeric concept**, the compiler tells you that

(aka does not satisfy

The messages also provide more detail, saying that

does not satisfy

and that

does not satisfy

A **static_assert** declaration optionally may specify as a second argument a string to include in the error message for `false` assertions.

15.8 Creating a Custom Algorithm

We saw in [Chapters 13](#) and [14](#) that the Standard Template Library is divided into containers, iterators and algorithms. We showed algorithms operating on container elements via iterators. You can take advantage of this architecture to **define custom algorithms capable of operating on any container that supports your algorithm's iterator requirements**. [Figure 15.10](#) defines a constrained average algorithm in which the argument must satisfy our custom `NumericInputRange` concept (described after the figure).

[Click here to view code image](#)

```
1 // fig15_10.cpp
2 // A custom algorithm to calculate the aver.
3 // a numeric input range's elements.
4 #include <algorithm>
```

```
5 #include <array>
6 #include <concepts>
7 #include <iostream>
8 #include <iterator>
9 #include <ranges>
10 #include <vector>
11
12 // concept for an input range containing integers
13 template<typename T>
14 concept NumericInputRange = std::ranges::input_range<
15     std::integral<typename T::value_type> | 
16     std::floating_point<typename T::value_type>
17
18 // calculate the average of a NumericInputRange
19 auto average(NumericInputRange auto const& range)
20     long double total{0};
21
22     for (auto i{range.begin()}; i != range.end(); ++i)
23         total += *i; // dereference iterator
24     }
25
26 // divide total by the number of elements
27 return total / std::ranges::distance(range);
28 }
29
30 int main() {
31     std::ostream_iterator<int> outputInt(std::cout);
32     const std::array ints{1, 2, 3, 4, 5};
33     std::cout << "array ints: ";
34     std::ranges::copy(ints, outputInt);
35     std::cout << "\n\naverage of ints: " << average();
36
37     std::ostream_iterator<double> outputDouble(std::cout);
38     const std::vector doubles{10.1, 20.2, 35.5};
39     std::cout << "\n\nvector doubles: ";
40     std::ranges::copy(doubles, outputDouble);
41     std::cout << "\n\naverage of doubles: " <<
```

```
42
43     std::ostream_iterator<long double> output;
44     const std::vector longDoubles{10.1L, 20.2L, 35.3L};
45     std::cout << "\n\nlist longDoubles: ";
46     std::ranges::copy(longDoubles, output);
47     std::cout << "\naverage of longDoubles: "
48             << "\n";
49 }
```



```
array ints: 1 2 3 4 5
average of ints: 3

vector doubles: 10.1 20.2 35.3
average of doubles: 21.8667

list longDoubles: 10.1 20.2 35.3
average of doubles: 21.8667
```

Fig. 15.10 | A custom algorithm to calculate the average of a numeric input range (Part 2 of 2.)

Concepts Custom NumericInputRange Concept

Lines 13–16 define the **custom NumericInputRange concept**, which checks:

- whether a type satisfies the `input_range` concept (line 14), so the argument supports **input iterators** for reading its elements, and
- whether range's elements satisfy the `std::integral` or `std::floating_point` concepts (lines 15–16), so they can be used in calculations.

Recall from [Section 13.2.1](#) that **each standard container has a nested value_type alias**, which indicates the container's element type. We use this to check whether the element type satisfies this concept's requirements. In lines 15 and 16, the notation

```
typename T::value_type
```

uses `typename` to indicate that `T::value_type` is an **alias for the range's element type**.

Custom average Algorithm

Lines 19–28 define `average` as an **abbreviated function template**. We constrained its `range` parameter with our **custom concept** `NumericInputRange`, so this algorithm can operate on any `input_range` of numeric values. To ensure that `average` can support any built-in numeric type, we use a `long double` (line 20) to store the sum of the elements. Lines 22–24 iterate through the `range` from its **begin iterator** up to, but not including, its **end iterator** and add each element's value to the total. Then line 27 divides the total by the range's number of elements, as determined by the `std::ranges::distance` algorithm.

Using Our average Algorithm on Standard Library Containers

To show that our custom `average` algorithm can process various standard library containers, lines 32, 38 and 44 define an array of `ints`, a `vector` of `doubles` and a `list` of `long doubles`, respectively. Lines 35, 41 and 47 call `average` with each of these containers to calculate the averages of their elements.

15.9 Creating a Custom Container and Iterators

 Chapter 11 introduced our `MyArray` class to demonstrate special member functions and operator overloading. The C++ Core Guidelines recommend that you implement as a template any class representing a container of values or range of values.⁶⁵ So, here, we'll implement a `MyArray` class template and enhance it using various STL conventions.⁶⁶ We'll also define **custom iterators** so `MyArray` objects can be used with various standard library algorithms.

65. “T.3: Use templates to express containers and ranges.” Accessed June 20, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-cont>.

66. Jonathan Boccaro, “Make Your Containers Follow the Conventions of the STL,” April 24, 2018.

Accessed May 24, 2021. <https://www.fluentcpp.com/2018/04/24/following-conventions-stl/>.

Our goal here is to give you a sense of what's involved in creating STL-like containers. Achieving full STL compatibility and backward compatibility with prior C++ language versions involves lots of **conditional compilation directives** beyond this book's scope. If you'd like to build reusable STL-like containers, study the code provided by the compiler vendors, and check out the research sources cited in our footnotes. Figure 15.11 defines our `MyArray` class template and its custom iterators. We broke the figure into several parts for discussion.

Single Header File

 One change you'll notice from Chapter 11's `MyArray` class is that **the entire class and its custom iterators are defined in a single header**, which is typical of class templates. **The compiler needs the complete definition where the template is used in order to specialize it.** Also, defining member functions inside the class template definition simplifies the syntax—you do not need `template` headers on each member function.

MyArray Supports Bidirectional Iterator

We modeled this example after the standard library's `array` class template, which uses a compile-time allocated, fixed-size, **built-in array**. As we discussed in Section 13.3, pointers into such arrays satisfy all the requirements of random-access iterators. For this example, however, we'll implement **custom bidirectional iterators** using class templates. The iterator architecture we use was inspired by the **Microsoft open-source C++ standard library `array`** implementation,⁶⁷ which **defines two iterator classes**:

⁶⁷. “array standard header.” Latest commit (i.e., when the file was last updated) February 24, 2021. Accessed May 24, 2021. <https://github.com/microsoft/STL/blob/main/stl/inc/array>.

- one for **iterators that manipulate `const` objects**, and
- one for **iterators that manipulate `non-const` objects**.

You can view their implementation at:

[Click here to view code image](#)

https://github.com/microsoft/STL/blob/main/stl/include/stl_iterator.h



We defined the following custom iterator classes:

- Our **ConstIterator** class represents **read-only bidirectional iterator**.
- Our **Iterator** class represents a **read/write bidirectional iterator**.

The **GNU** and **Clang** array implementations simply use pointers for their **array** iterators. You can see their implementations at

[Click here to view code image](#)

<https://github.com/gcc-mirror/gcc/blob/master/libc/include/std/array>



and

[Click here to view code image](#)

<https://github.com/llvm/llvm-project/blob/main/include/llvm/Support/Container.h>



Why We Implemented Bidirectional Rather Than Random-Access Iterators

Recall from our introduction to iterators in [Section 13.3](#) that various levels of iterators are supported by STL containers. The most powerful are **random-access iterators**. Most standard library algorithms operate on ranges of container elements, and **only 12 algorithms require random-access iterators**—shuffle and 11 sorting-related algorithms. **MyArray's bidirectional iterators enable most standard library algorithms to process MyArray objects.** Random-access iterators have many additional requirements⁶⁸ shown in the table below. As an exercise, you could enhance MyArray's custom iterators with these capabilities to make them **random-access iterators**.

68. “23.3.4.13 Concept random_access_iterator [iterator.concept.random.access].” Accessed June 27, 2021. <https://eel.is/c++draft/iterator.concept.random.access>.

Random-access iterator operation	Description
<code>p += i</code>	Increment the iterator <code>p</code> by <code>i</code> positions.
<code>p -= i</code>	Decrement the iterator <code>p</code> by <code>i</code> positions.
<code>p + i or i + p</code>	Result is an iterator positioned at <code>p</code> incremented by <code>i</code> positions.
<code>p - i</code>	Result is an iterator positioned at <code>p</code> decremented by <code>i</code> positions.
<code>p - p1</code>	Calculate the distance (that is, number of elements) between two elements in the same container.
<code>p[i]</code>	Return a reference to the element offset from <code>p</code> by <code>i</code> positions
<code>p < p1</code>	Return <code>true</code> if iterator <code>p</code> is less than iterator <code>p1</code> (that is, iterator <code>p</code> is before iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p <= p1</code>	Return <code>true</code> if iterator <code>p</code> is less than or equal to iterator <code>p1</code> (that is, iterator <code>p</code> is before or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p > p1</code>	Return <code>true</code> if iterator <code>p</code> is greater than iterator <code>p1</code> (that is, iterator <code>p</code> is after iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p >= p1</code>	Return <code>true</code> if iterator <code>p</code> is greater than or equal to iterator <code>p1</code> (that is, iterator <code>p</code> is after or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .

Basic Iterator Requirements

All iterators must support.^{69,70,71}

69. “C++ named requirements: LegacyIterator,” Accessed May 24, 2021. https://en.cppreference.com/w/cpp/named_req/Iterator.
70. Triangles, “Writing a custom iterator in modern C++,” December 19, 2020. Accessed May 24, 2021. <https://internalpointers.com/post/writing-custom-iterators-modern-cpp>.
71. David Gorski, "Custom STL Compatible Iterators," March 2, 2019. Accessed May 24, 2021. <https://davidgorski.ca/posts/stl-iterators/>.

- default construction,

- copy construction,
- copy assignment,
- destruction, and
- swapping.

We implement our iterator classes using the “**Rule of Zero**” (Section 11.6.6), letting the compiler generate the **copy constructor**, **copy assignment operator**, **move constructor**, **move assignment operator** and **destructor** special member functions.

15.9.1 Class Template ConstIterator

Lines 14–76 of Fig. 15.11 define the **class template ConstIterator**, which class MyArray uses to create **read-only iterators**. The template has one type parameter T (line 14), representing a MyArray’s element type. Each ConstIterator points to an element of that type.

[Click here to view code image](#)

```

1 // Fig. 15.11: MyArray.h
2 // Class template MyArray with custom iterator
3 // by class templates ConstIterator and Iterator
4 #pragma once
5 #include <algorithm>
6 #include <initializer_list>
7 #include <iostream>
8 #include <iterator>
9 #include <memory>
10 #include <stdexcept>
11 #include <utility>
12
13 // class template ConstIterator for a MyArray
14 template <typename T>
15 class ConstIterator {
16 public:
17     // public iterator nested type names
18     using iterator_category = std::bidirectional_iterator_tag;

```

```
19     using difference_type = std::ptrdiff_t;
20     using value_type = T;
21     using pointer = const value_type*;
22     using reference = const value_type&;
23
24     // default constructor
25     ConstIterator() : m_ptr{nullptr} {}
26
27     // initialize a ConstIterator with a pointer
28     ConstIterator(pointer p) : m_ptr{p} {}
29
30     // OPERATIONS ALL ITERATORS MUST PROVIDE
31     // increment the iterator to the next element
32     // return a reference to the iterator
33     ConstIterator& operator++() noexcept {
34         ++m_ptr;
35         return *this;
36     }
37
38     // increment the iterator to the next element
39     // return the iterator before the increment
40     ConstIterator operator++(int) noexcept {
41         ConstIterator temp{*this};
42         ++(*this);
43         return temp;
44     }
45
46     // OPERATIONS INPUT ITERATORS MUST PROVIDE
47     // return a const reference to the element
48     reference operator*() const noexcept { re
49
50     // return a const pointer to the element
51     pointer operator->() const noexcept { ret
52
53     // <=> operator automatically supports equality
54     // Only == and != are needed for bidirectional
55     // This implementation would support the
```

```

56     // by random-access iterators.
57     auto operator<=>(const ConstIterator& ot:
58
59     // OPERATIONS BIDIRECTIONAL ITERATORS MU
60     // decrement the iterator to the previou
61     // return a reference to the iterator
62     ConstIterator& operator--() noexcept {
63         --m_ptr;
64         return *this;
65     }
66
67     // decrement the iterator to the previou
68     // return the iterator before the deccre
69     ConstIterator operator--(int) noexcept {
70         ConstIterator temp{*this};
71         --(*this);
72         return temp;
73     }
74     private:
75         pointer m_ptr;
76     };
77

```



Fig. 15.11 | Class template MyArray with custom iterators implemented by class templates ConstIterator and Iterator—class ConstIterator. (Part 3 of 3.)

Standard Iterator Nested Type Names

Lines 18–22 define type aliases for the **nested type names that the C++ standard library expects in iterator classes**:⁷²

72. “23.3.2.3 Iterator traits.” Accessed June 30, 2021.
<https://eel.is/c++draft/iterators#iterator.traits>.

- **20 iterator_category:** The iterator’s category (Section 13.3.2), specified here as the type `std::bidirectional_iterator_tag` from header `<iterator>`. This “tag” type indicates **bidirectional**

iterators. The standard library algorithms use type traits and C++20 concepts to confirm that a container’s iterators have the correct category for use with each algorithm.

- **difference_type:** The result type of subtracting one ConstIterator from another—`std::ptrdiff_t` is the result type for pointer subtraction.
- **value_type:** The element type to which a ConstIterator points.
- **pointer:** The type of a pointer to a const object of the value_type. The Const-Iterator’s data member (line 75) is declared with this type.
- **reference:** The type of a reference to a const object of the value_type.

Constructors

Class ConstIterator provides a no-argument constructor that initializes a ConstIterator to `nullptr` (line 25), and a constructor that initializes a ConstIterator from a pointer to an element (line 28). **The class’s copy and move constructors are autogenerated.**

++ Operators

Lines 33–36 and 40–44 define the preincrement and postincrement operators required by all iterators. The preincrement operator aims the iterator’s `m_ptr` member at the next element and returns a reference to the incremented iterator. The postincrement operator aims the iterator’s `m_ptr` member at the next element and returns a copy of the iterator before the increment.

Overloaded * and -> Operators

Semantically, iterators are like pointers, so they must overload the `*` and `->` operators (lines 48 and 51). The overloaded `operator*` dereferences `m_ptr` to access the element the iterator currently points to and returns a reference to that element. The overloaded `operator->` returns `m_ptr` as a pointer to that element.

Bidirectional Iterator Comparisons

Bidirectional iterators must be comparable with == and !=. Here we used the **compiler-generated three-way comparison operator <=>** (line 57) to support ConstIterator comparisons. If you enhance our iterator classes to make them random-access iterators, this implementation also enables comparisons with the operators <, <=, > and >=, as required by random-access iterators.

-- Operators

Lines 62–65 and 69–73 define the predecrement and postdecrement operators required by bidirectional iterators. These operators work like the ++ operators, but aim the m_ptr member at the previous element.

15.9.2 Class Template Iterator

Lines 80–136 of Fig. 15.11 define the **class template Iterator**, which class MyArray uses to create **read/write iterators**. The template has one type parameter T (line 80), representing a MyArray's element type. Class Iterator inherits from class ConstIterator<T> (line 81).

[Click here to view code image](#)

```
78 // class template Iterator for a MyArray no:
79 // redefines several inherited operators to
80 template <typename T>
81 class Iterator : public ConstIterator<T> {
82 public:
83     // public iterator nested type names
84     using iterator_category = std::bidirecti-
85     using difference_type = std::ptrdiff_t;
86     using value_type = T;
87     using pointer = value_type*;
88     using reference = value_type&;
89
90     // inherit ConstIterator constructors
91     using ConstIterator<T>::ConstIterator;
92
93     // OPERATIONS ALL ITERATORS MUST PROVIDE
```

```
94     // increment the iterator to the next element
95     // return a reference to the iterator
96     Iterator& operator++() noexcept {
97         ConstIterator<T>::operator++(); // call base class
98         return *this;
99     }
100
101    // increment the iterator to the next element
102    // return the iterator before the increment
103    Iterator operator++(int) noexcept {
104        Iterator temp{*this};
105        ConstIterator<T>::operator++(); // call base class
106        return temp;
107    }
108
109    // OPERATIONS INPUT ITERATORS MUST PROVIDE
110    // return a const reference to the element
111    // operator returns a non-const reference
112    reference operator*() const noexcept {
113        return const_cast<reference>(ConstItera-
114    }
115
116    // return a const pointer to the element
117    pointer operator->() const noexcept {
118        return const_cast<pointer>(ConstItera-
119    }
120
121    // OPERATIONS BIDIRECTIONAL ITERATORS MUST
122    // decrement the iterator to the previous element
123    // return a reference to the iterator
124    Iterator& operator--() noexcept {
125        ConstIterator<T>::operator--(); // call base class
126        return *this;
127    }
128
129    // decrement the iterator to the previous element
130    // return the iterator before the decrement
```

```

131     Iterator operator--(int) noexcept {
132         Iterator temp{*this};
133         ConstIterator<T>::operator--(); // ca
134         return temp;
135     }
136 };
137

```



Fig. 15.11 | Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `Iterator`. (Part 2 of 2.).

Standard Iterator Nested Type Names

Lines 84–88 define **type aliases for the nested type names that the C++ standard library expects in iterator classes**. Class template `Iterator` defines **read/write iterators**, so lines 87–88 define the pointer and reference **type aliases** without `const`. Pointers and references of these types can be used to write new values into elements.

Constructors

`ConstIterator`'s constructors know how to initialize `Iterator`'s inherited `m_ptr` member, so line 91 simply inherits the base class's constructors.

++ and -- Operators

Lines 96–99, 103–107, 124–127 and 131–135 define the preincrement, postincrement, predecrement and postdecrement operators. Each simply calls `ConstIterator`'s corresponding version. The prefix operators return a reference to the updated `Iterator`. The postfix operators return a copy of the `Iterator` before the increment or decrement.

Overloaded * and -> Operators

Lines 112–114 and 117–119 overload the `*` and `->` operators. Each calls `ConstIterator`'s version. Those versions return a pointer or reference that views the `value_type` as `const`. Since `Iterators` should allow

both reading and writing element values, lines 113 and 118 use `const_cast` to **cast away the const-ness** (that is, remove the `const`) of the pointer or reference returned by the base-class overloaded operators.

15.9.3 Class Template MyArray

Lines 140–210 of Fig. 15.11 define our simplified MyArray class template. We removed some overloaded operators and various special member functions presented in Chapter 11 to focus on the **container** and its **iterators**. To mimic the `std::array` class template, we define MyArray as an **aggregate type** (Section 9.21), which requires all non-static data to be `public`. So, we defined MyArray using a struct, which has `public` members by default. Also, rather than using dynamic memory allocation, the new MyArray class template's data is stored as a fixed-size built-in array (line 209) allocated at compilation time.

[Click here to view code image](#)

```
138 // class template MyArray contains a fixed
139 // MyArray is an aggregate type with public
140 template <typename T, size_t SIZE>
141 struct MyArray {
142     // type names used in standard library
143     using value_type = T;
144     using size_type = size_t;
145     using difference_type = ptrdiff_t;
146     using pointer = value_type*;
147     using const_pointer = const value_type*;
148     using reference = value_type&;
149     using const_reference = const value_type&;
150
151     // iterator type names used in standard
152     using iterator = Iterator<T>;
153     using const_iterator = ConstIterator<T>;
154     using reverse_iterator = std::reverse_it
155     using const_reverse_iterator = std::rev
156
```

```
157 // Rule of Zero: MyArray
158
159 constexpr size_type size() const noexcept
160
161 // member functions that return iterator
162 iterator begin() {return iterator{&m_data[0]};}
163 iterator end() {return iterator{&m_data[SIZE]};}
164 const_iterator begin() const {return const_iterator{&m_data[0]};}
165 const_iterator end() const {
166     return const_iterator{&m_data[0] + size()};
167 }
168 const_iterator cbegin() const {return begin();}
169 const_iterator cend() const {return end();}
170
171 // member functions that return reverse iterator
172 reverse_iterator rbegin() {return reverse_iterator{&m_data[SIZE]};}
173 reverse_iterator rend() {return reverse_iterator{&m_data[0]};}
174 const_reverse_iterator rbegin() const {
175     return const_reverse_iterator{cend()};
176 }
177 const_reverse_iterator rend() const {
178     return const_reverse_iterator{cbegin()};
179 }
180 const_reverse_iterator crbegin() const {
181     return const_reverse_iterator{rbegin()};
182 }
183 // auto-generated three-way comparison operators
184 auto operator<=(const MyArray& t) const {
185
186 // overloaded subscript operator for non-const objects
187 // reference return creates a modifiable copy
188 T& operator[](size_type index) {
189     // check for index out-of-range error
190     if (index >= SIZE) {
191         throw std::out_of_range("Index out of range");
192     }
193 }
```

```

194         return m_data[index]; // reference r
195     }
196
197     // overloaded subscript operator for co
198     // const reference return creates a non-
199     const T& operator[](size_type index) co
200         // check for subscript out-of-range
201         if (index >= SIZE) {
202             throw std::out_of_range("Index ou
203         }
204
205         return m_data[index]; // returns cop
206     }
207
208     // like std::array the data is public t
209     T m_data[SIZE]; // built-in array of ty
210 }
211

```



Fig. 15.11 | Class template MyArray with custom iterators implemented by class templates ConstIterator and Iterator—class MyArray. (Part 2 of 2.)

MyArray's template Header

The template header (line 140) indicates that MyArray is a **class template**. The header specifies two parameters:

- T represents the MyArray's element type.
- SIZE is a **non-type template parameter** that's treated as a compile-time constant. We use SIZE to represent the MyArray's number of elements.

Line 209 creates a built-in array of type T containing SIZE elements. Though we do not do so here, non-type template parameters can have **default arguments**.

Standard Container Nested Type Names

Lines 143–155 define type aliases for the nested type names that the C++ standard library expects in container classes. The types for **reverse iterators** are specific to containers with **bidirectional** or **random-access iterators**:⁷³

73. “Table 73: Container requirements [tab:container.req].” Accessed June 30, 2021. <https://eel.is/c++draft/container.requirements#tab:container.req>.

- **value_type**: The container’s element type (`T`).
- **size_type**: The type used representing the container’s number of elements.
- **difference_type**: The result type when subtracting iterators.
- **pointer**: The type of a pointer to a `value_type` object.
- **const_pointer**: The type of a pointer to a `const value_type` object.
- **reference**: The type of a reference to a `value_type` object.
- **const_reference**: The type of a reference to a `const value_type` object.
- **iterator**: `MyArray`’s read/write iterator type (`Iterator<T>`).
- **const_iterator**: `MyArray`’s read-only iterator type (`ConstIterator<T>`).
- **reverse_iterator**: `MyArray`’s read/write iterator type for iterating backward from the end of a `MyArray`. The **iterator adapter** `std::reverse_iterator` creates a reverse-iterator type from its bidirectional or random-access iterator type argument.
- **const_reverse_iterator**: `MyArray`’s read-only iterator type for moving backward from the end of the `MyArray` —`std::reverse_iterator` creates a `const` reverse-iterator type from its bidirectional or random-access iterator type argument.

MyArray Member Functions That Return Iterators

Lines 162–181 define the `MyArray` member functions that return `MyArray`’s various kinds of iterators and reverse iterators. The key functions are `begin` and `end` in lines 162–167:

- **begin** (line 162) returns an `iterator` pointing to the `MyArray`’s

first element.

- **end** (line 163) returns an iterator pointing to the MyArray’s last element.
- **begin** (line 164) is an overload that returns a `const_iterator` pointing to the MyArray’s first element.
- **end** (line 165–167) is an overload that returns a `const_iterator` pointing to the MyArray’s last element.

The other member functions that return iterators call these `begin` and `end` functions:

- **cbegin** and **cend** (lines 168–169) call the versions of `begin` and `end` that return `const` iterators.
- **rbegin** and **rend** (lines 172–173) produce `reverse_iterators` based on the iterators returned by the **non-const versions of begin and end**.
- **rbegin** and **rend** (lines 174–179) are overloads that produce `const_reverse_iterators` based on the `const_iterators` returned by the **const versions of begin and end**.
- **crbegin** and **crend** (lines 180–181) return the `const_reverse_iterators` from calling the `const` versions of `rbegin` and `rend`.

MyArray Overloaded Operators

Lines 184, 188–195 and 199–206 define MyArray’s overloaded operators. Though we won’t use it in this example, the **compiler-generated three-way comparison operator** (line 184) enables you to compare entire MyArray objects of the same specialized type. The overloaded operator `[]` member functions are identical to those in [Chapter 11](#)’s MyArray class, but we now use the **non-type template parameter SIZE** (lines 190 and 201) when determining whether an `index` is outside a MyArray’s bounds.

15.9.4 MyArray Deduction Guide for Braced Initialization

As you’ve seen, you can initialize a `std::array` using **class-template argument deduction (CTAD)**. For example, when the compiler sees the

statement:

[Click here to view code image](#)

```
std::array ints{1, 2, 3, 4, 5};
```

it infers that the array's element type is `int` because all the initializers are `ints`, and it counts the initializers to determine the array's size. For our `MyArray` class template, we did not define a constructor that can receive any number of arguments. However, we can define a **deduction guide**⁷⁴ (lines 213–214) that shows the compiler how to deduce a `MyArray`'s type from a braced initializer. Then the compiler can use **aggregate initialization** to place the initializers into our `MyArray` aggregate type's built-in array data member.

74. "Class template argument deduction." Accessed June 30, 2021.
https://en.cppreference.com/w/cpp/language/class_template_argument

[Click here to view code image](#)

```
212 // deduction guide to enable MyArrays to b...
213 template<typename T, typename... U>
214 MyArray(T first, U... rest) -> MyArray<T,
```



Fig. 15.11 | Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—`MyArray` deduction guide.

A deduction guide is a **template**. This deduction guide's template header uses **variadic template syntax** (...), which we discuss extensively in Section 15.15. Line 213 indicates that the compiler is looking for an initializer list that

- contains at least one initializer, specified by `typename T`, and
- may contain any number of additional initializers, as specified by `typename... U`, which is known as a **parameter pack**.

To the left of the `->` in line 214, you specify what looks like the beginning of

a MyArray constructor that receives its first argument in the T parameter named `first` and all other arguments in the U... parameter named `rest`. To the right of the -> you tell the compiler that when it sees a MyArray initialized using **class template argument deduction**, it should deduce that we want to create a MyArray with elements of type T and with its size specified by

```
1 + sizeof... (U)
```

In this expression,

- 1 is the initializer list's minimum number of initializers and
- `11 sizeof... (U)` uses C++11's compile-time **sizeof...** **operator** to determine the additional number of initializers the compiler placed in the parameter pack U.

Our deduction guide is a simplified version of those provided by GNU and Clang for their `std::array` implementations. Their deduction guides also ensure that all the initializers have the same type. You can view GNU's and Clang's deduction guides in their respective `<array>` headers at:

[Click here to view code image](#)

```
https://github.com/gcc-mirror/gcc/blob/master/libc++  
include/std/array
```

and

[Click here to view code image](#)

```
https://github.com/llvm/llvm-project/blob/main/include  
clang/libc++/array.h
```

15.9.5 Using MyArray and Its Custom Iterators with std::ranges Algorithms

Figure 15.12 creates three MyArrays that store ints, doubles and strings, respectively, then uses them with various `std::ranges` algorithms that require `input`, `output`, `forward` or `bidirectional iterators`. We also use a `range-based for statement` to iterate through a `MyArray`.

MyArray has **bidirectional iterators**, so we also could use it with our custom average algorithm in Fig. 15.10, which required only **input iterators**.

[Click here to view code image](#)

```
1 // fig15_12.cpp
2 // Using MyArray with range-based for and w
3 // C++ standard library algorithms.
4 #include <iostream>
5 #include <iterator>
6 #include "MyArray.h"
7
8 int main() {
9     std::ostream_iterator<int> outputInt{std::cout};
10    std::ostream_iterator<double> outputDoub
11    std::ostream_iterator<std::string> outpu
12
13    std::cout << "Displaying MyArrays with s
14        << "which requires input iterators:\n
15    MyArray ints{1, 2, 3, 4, 5, 6, 7, 8};
16    std::cout << "ints: ";
17    std::ranges::copy(ints, outputInt);
18
19    MyArray doubles{1.1, 2.2, 3.3, 4.4, 5.5}
20    std::cout << "\ndoubles: ";
21    std::ranges::copy(doubles, outputDouble)
22
23    MyArray strings{"red", "orange", "yellow",
24        std::cout << "\nstrings: ";
25    std::ranges::copy(strings, outputString)
26
27    std::cout << "\n\nDisplaying a MyArray w
28        << "statement, which requires input i
29    for (const auto& item : doubles) {
30        std::cout << item << " ";
```

```
31     }
32
33     std::cout << "\n\nCopying a MyArray with
34         << "which requires input iterators:\n"
35     MyArray<std::string, strings.size()> str
36     std::ranges::copy(strings, strings2.begin());
37     std::cout << "strings2 after copying from
38     std::ranges::copy(strings2, outputString
39
40     std::cout << "\n\nFinding min and max el-
41         << "with std::ranges::minmax_element,
42             << "forward iterators:\n";
43     const auto& [min, max] {std::ranges::minmax_
44     std::cout << "min and max elements of st
45         << *min << ", " << *max;
46
47     std::cout << "\n\nReversing a MyArray wi
48         << "which requires bidirectional iter
49     std::ranges::reverse(ints);
50     std::cout << "ints after reversing elemen
51     std::ranges::copy(ints, outputInt);
52     std::cout << "\n";
53 }
```

< >

```
Displaying MyArrays with std::ranges::copy, which
ints: 1 2 3 4 5 6 7 8
doubles: 1.1 2.2 3.3 4.4 5.5
strings: red orange yellow
```

```
Displaying a MyArray with a range-based for stat
iterators:
1.1 2.2 3.3 4.4 5.5
```

```
Copying a MyArray with std::ranges::copy, which
strings2 after copying from strings1: red orange
```

```
Finding min and max elements in a MyArray with std::min_element and std::max_element, which requires forward iterators:  
min and max elements of strings are: orange, yellow  
  
Reversing a MyArray with std::ranges::reverse, which requires bidirectional iterators:  
ints after reversing elements: 8 7 6 5 4 3 2 1
```

Fig. 15.12 | Using MyArray with range-based `for` and with C++ standard library algorithms. (Part 3 of 3.)

Creating MyArrays and Displaying Them with `std::ranges::copy`

In lines 13–25, we create three MyArrays (lines 15, 19 and 23), using **class template argument deduction** to infer their element types and sizes. Lines 17, 21 and 25 display each MyArray’s contents using `std::ranges::copy`. This algorithm’s first argument is an **input_range** which requires the range to have **input iterators**, so MyArrays are compatible with this algorithm because they have more powerful **bidirectional iterators**.

Displaying a MyArray with a Range-Based `for` Statement

17 Lines 29–31 display the contents of MyArray doubles using a range-based `for`, which requires only **input iterators**, so it **works with any iterable object**, including MyArrays. In this case, the MyArray doubles is not a `const` object, so the MyArray’s read/write iterators are used. **If you have a non-const object that you want to treat as const, you can create a const view of the non-const object by passing it to C++17’s `std::as_const` function.** For example, this example’s range-based `for` does not modify doubles’ elements, so we could have written line 29 as

[Click here to view code image](#)

```
for (auto& item : std::as_const(doubles)) {
```

In this case, `item`’s type will be inferred as a reference to a `const`

double element.

Copying a MyArray with `std::ranges::copy`

Line 35 creates a new `MyArray` into which we'll copy the `MyArray` `strings`' elements. Line 36 uses `std::ranges::copy` to copy `strings`' elements into `strings2`. The algorithm uses an **input iterator** to read each element from `strings` and an **output iterator** to specify where to write the element into `strings2`. `MyArray`'s **non-const bidirectional iterators** support both reading (input) and writing (output), so one `MyArray` can be copied into another of the same specialized type with `std::ranges::copy`.

Finding the Minimum and Maximum Elements in a MyArray with `std::ranges::minmax_element`

Line 43 uses the `std::ranges::minmax_element` algorithm to get iterators pointing to the elements containing a `MyArray`'s minimum and maximum values. This algorithm requires a **forward range**, which provides **forward iterators**. `MyArray`'s **bidirectional iterators** are more powerful than **forward iterators**, so the algorithm can operate on a `MyArray`.

Copying a MyArray with `std::ranges::copy`

Line 49 reverses `MyArray` `ints`' elements using the `std::ranges::reverse` algorithm, which requires a **bidirectional_range** with **bidirectional iterators**. These are the exact iterators provided by `MyArray`, so the algorithm can operate on a `MyArray`.

Attempting to Use MyArray with `std::ranges::sort`

 `MyArray`'s iterators do not support all the features of **random-access iterators**, so we **cannot pass a MyArray to algorithms that require them**. To prove this, we wrote a short program containing only the following statements that create a `MyArray` of `ints` then attempt to sort it with `std::ranges::sort`, which requires **random-access iterators**:

[Click here to view code image](#)

```
MyArray integers{10, 2, 33, 4, 7, 1, 80};  
std::ranges::sort(integers);
```

The **Clang** compiler produced the error messages in the output window below. We highlighted key messages in bold and added some blank lines for readability. The messages clearly indicate that the code does not compile

because

and, in turn,

because not satisfy

[Click here to view code image](#)

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../.../.../  
es_algo.h:2030:7: note: candidate template ignored  
[with _Range = MyArray<int, 7> &, _Comp = std::range  
std::identity]  
    operator()(_Range&& __r, _Comp __comp = {}, -  
    ^  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../.../.../  
es_algo.h:2026:14: note: because dom_access_range  
    template<random_access_range _Range,  
    ^  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../.../.../  
range_access.h:924:37: note: because erator<int>  
    = bidirectional_range<_Tp> && random_access_<  
    ^  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../.../.../  
ator_concepts.h:591:10: note: because cept<Iterator  
cess_iterator_tag>  
    && derived_from<__detail::__iter_concept<_It  
    ^  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../.../...
```

```
cepts:67:28: note: because std::bidirectional_iterator
concept derived_from = __is_base_of(_Base, _Dei
                                ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../...
rang8least 2 arguments, but 1 was provided
        operator()(_Iter __first, _Sent __last,
        ^
1 error generated.
```

15.10 Default Arguments for Template Type Parameters

A type parameter also can specify a **default type argument**. For example, the C++ standard's **stack container adapter** class template begins with:

[Click here to view code image](#)

```
template <class T, class Container = deque<T>>
```

which specifies that a **stack** uses a **deque** by default to store the **stack**'s elements of type **T**. When the compiler sees the declaration

```
stack<int> values;
```

it **specializes stack for type int** and uses the specialization to instantiate the object named **values**. The **stack**'s elements are stored in a **deque<int>**.

11 Default type parameters must be the rightmost (trailing) parameters in a template's type-parameter list. When you instantiate a template with two or more default arguments, if an omitted argument is not the rightmost, all type parameters to the right of it also must be omitted. **C++11 added the ability to use default type arguments for template type parameters in function templates.**

15.11 Variable Templates

14 You've specialized function templates and class templates to define groups of related functions and classes, respectively. C++14 added **variable templates**, which can be specialized to define groups of related variables. **When placed at class scope** (inside a class but outside its member functions), **variable templates define static class variables**.

17 You used several **predefined variable templates** in Fig. 15.6's type traits demonstration. C++17 added convenient **variable templates for accessing each type trait's value member**. For example, rather than testing whether a type T is an integral type with:

```
is_integral<T>::value
```

we can use the corresponding **variable template**:

```
is_integral_v<T>
```

which is defined in the standard as:

[Click here to view code image](#)

```
template<class T>
inline constexpr bool is_arithmetic_v = is_arithme
< >
```

17 Err~~⊗~~ SE A This variable template defines an `inline bool` variable that evaluates to a **compile-time constant** (indicated by `constexpr`). C++17 added `inline` variables to better support header-only libraries, which can be included in multiple source-code files—known as **compilation units**—within the same application.⁷⁵ When you define a regular variable in a header, including that header more than once in an application results in **multiple definition errors** for that variable. On the other hand, **identical inline variable definitions are allowed in separate compilation units within the same application.**⁷⁶

⁷⁵. Alex Pomeranz, “6.8 — Global constants and inline variables,” January 3, 2020. Accessed June 13, 2021. <https://www.learncpp.com/cpp-tutorial/global-constants-and-inline-variables/>.

⁷⁶. “`inline` specifier.” Accessed June 24, 2021. <https://en.cppreference.com/w/cpp/language/inline>.

15.12 Variadic Templates and Fold Expressions

11 SE  Before C++11, each class template or function template had a fixed number of template parameters. Defining a class or function template with **different numbers of template parameters** required a separate template definition for each case. C++11 **variadic templates accept any number of arguments**. They simplify template programming because you can provide one variadic function template rather than many overloaded ones with different numbers of parameters.

15.12.1 `tuple` Variadic Class Template

11 C++11's `tuple` class (from header `<tuple>`) is a **variadic-class-template generalization of class template pair** (introduced in [Section 13.9](#)). A `tuple` is a collection of related values, possibly of mixed types. [Figure 15.13](#) demonstrates several `tuple` capabilities.

[Click here to view code image](#)

```
1 // fig15_13.cpp
2 // Manipulating tuples.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6 #include <tuple>
7
8 // type alias for a tuple representing a ha
9 using Part = std::tuple<int, std::string, i:
10
11 // return a part's inventory tuple
12 Part getInventory(int partNumber) {
13     using namespace std::string_literals; //
14
15     switch (partNumber) {
16     case 1:
17         return {1, "Hammer"s, 32, 9.95}; // r
```

```
18     case 2:
19         return {2, "Screwdriver"s, 106, 6.99}
20     default:
21         return {0, "INVALID PART"s, 0, 0.0};
22     }
23 }
24
25 int main() {
26     // display the hardware part inventory
27     for (int i{1}; i <= 2; ++i) {
28         // unpack the returned tuple into four
29         // variables' types are inferred from
30         auto [partNumber, partName, quantity,
31
32             std::cout << fmt::format("{}: {}, {}:
33                 "Part number", partNumber, "Tool",
34                 "Quantity", quantity, "Price", pri
35             }
36
37             std::cout << "\nAccessing a tuple's elem
38             auto hammer{getInventory(1)};
39             std::cout << fmt::format("{}: {}, {}: {}
40                 "Part number", std::get<0>(hammer), ""
41                 "Quantity", std::get<2>(hammer), "Pri
42
43             std::cout << fmt::format("A Part tuple h
44             std::tuple_size<Part>{}));
45 }
```

```
< >
Part number: 1, Tool: Hammer, Quantity: 32, Price: 6.99
Part number: 2, Tool: Screwdriver, Quantity: 106, Price: 6.99

Accessing a tuple
Part number: 1, Tool: Hammer, Quantity: 32, Price: 6.99
A Part tuple has 4 elements
```

Fig. 15.13 | Manipulating tuples. (Part 2 of 2.)

Line 9's using declaration (introduced in [Section 10.13](#)) creates a **type alias** named `Part` for a `tuple` representing a hardware part's inventory. Each `Part` consists of

- an `int` part number,
- a `string` part name,
- an `int` quantity and
- a `double` price.

In a `tuple` declaration, every template type parameter corresponds to a value at the same position in the `tuple`. **The number of tuple elements always matches the number of type parameters.** We use the `Part` type alias to simplify the rest of the code.

Packing a tuple

Creating a `tuple` object is called **packing a tuple**. Lines 12–23 define a `getInventory` function that receives a part number and returns a `Part` tuple. The function packs `Part` tuples by returning an **initializer list** (lines 17, 19 or 21) containing four elements (an `int`, a `string`, an `int` and a `double`), which the compiler uses to initialize the returned `Part std::tuple` object. A `tuple`'s size is fixed once you create it, so `tuples` are said to be immutable.

Creating a tuple with `std::make_tuple`

You also can pack a `tuple` with the `<tuple>` header's **`make_tuple` function**, which infers a `tuple`'s type parameters from the function's arguments. For example, you could create the `tuples` in lines 17, 19 and 21 with `make_tuple` calls like

[Click here to view code image](#)

```
std::make_tuple(1, "Hammer"s, 32, 9.95)
```

If we had used this approach, `getInventory`'s return type could be specified as **`auto` to infer the return type from `make_tuple`'s result.**

Unpacking a tuple with Structured Bindings

You can **unpack a tuple** to access its elements using **C++17 structured bindings** (Section 14.4.3). Line 30 unpacks a Part’s members into variables, which we display in lines 32–34.

Using `get<index>` to Obtain a tuple Member by Index

Class template pair contains public members first and second for accessing a pair’s two members. Each tuple you create can have any number of elements, so tuples do not have similarly named data members. Instead, the `<tuple>` header provides the function template `get< index>(tupleObject)`, which returns a reference to the *tupleObject*’s member at the specified *index*. The first member has index 0. Line 38 gets a tuple for the hammer inventory, then lines 39–41 access each tuple element by index.

C++14 Using `get< type>` to Obtain a tuple Member By Type

14 C++14 added a get overload that gets a tuple member of a specific type, provided that the tuple contains **only one member of that type**. For example, the following statement gets the hammerInventory tuple’s string member:

[Click here to view code image](#)

```
auto partName{get<std::string>(hammerInventory)};
```

However, the statement

[Click here to view code image](#)

```
auto partNumber{get<int>(hammerInventory)};
```

 would generate a compilation error because the call is **ambiguous**—the hammerInventory tuple contains two int members for its part number and quantity.

Other tuple Features

The following table shows several other **tuple class template features**. For the tuple class template’s complete details, see:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/utility/tuple/t>

< >

For other `tuple`-related utilities defined in the `<tuple>` header, see:

[Click here to view code image](#)

<https://en.cppreference.com/w/cpp/utility/tuple>

Other <code>tuple</code> class template features	Description
comparisons	<code>tuples</code> that contain the same number of members can be compared to one another using the relational and equality operators. Corresponding members of each <code>tuple</code> are compared and must support <code><</code> to work with a <code>tuple</code> 's relational operators or <code>==</code> to work with a <code>tuple</code> 's equality operators.
default constructor	Creates a <code>tuple</code> in which each member is value initialized. Primitive type values are set to 0 or the equivalent of 0. Objects of class types are initialized with their default constructors.
copy constructor	Copies a <code>tuple</code> 's elements into a new <code>tuple</code> of the same type. The element types stored in the constructor argument must have a copy constructor.
move constructor	Moves a <code>tuple</code> 's elements into a new <code>tuple</code> of the same type.
copy assignment	Uses the assignment operator (<code>=</code>) to copy <code>tuple</code> elements in the right operand into a <code>tuple</code> of the same type in the left operand. The element types stored in the constructor argument must be copy assignable.
move assignment	Uses the assignment operator (<code>=</code>) to move <code>tuple</code> elements in the right operand into a <code>tuple</code> of the same type in the left operand. The element types stored in the constructor argument must be move assignable.

15.12.2 Variadic Function Templates and an Intro to C++17 Fold Expressions

Variadic function templates enable you to define functions that can receive any number of arguments. Figure 15.14 uses **variadic function templates** to sum one or more arguments. This example assumes the arguments can be operands to the `+` operator and have the same type. We show two ways to process the variadic parameters:

- using **compile-time recursion** (which was required before C++17), and
 - using a **C++17 fold expression** to eliminate the recursion.

[Section 15.12.7](#) shows how to **test whether all the arguments have the same type**.

[Click here to view code image](#)

```

30      "sum(1): ", sum(1), "sum(1, 2): "
31      "sum(1, 2, 3): ", sum(1, 2, 3),
32      "sum(\"s\"s, \"u\"s, \"m\"s): "
33
34      std::cout << "Variadic function template "
35      << fmt::format("\n{}{}\n{}{}\n{}{}\n{
36          "sum(1): ", foldingSum(1), "sum"
37          "sum(1, 2, 3): ", foldingSum(1,
38          "sum(\"s\"s, \"u\"s, \"m\"s): "
39          foldingSum("s"s, "u"s, "m"s));
40 }

```



Recursive variadic function template sum:

```

sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

```

Variadic function template foldingSum:

```

sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

```

Fig. 15.14 | Variadic function templates. (Part 2 of 2.)

Compile-Time Recursion

Lines 8–11 and 14–17 define overloaded function templates named `sum` that use **compile-time recursion** to process **variadic parameter packs**. The function template `sum` with one template parameter (lines 8–11) represents the **recursion's base case** in which `sum` receives only one argument and returns it. The **recursive function template `sum`** (lines 14–17) specifies two type parameters:

- The type parameter `FirstItem` represents the first function argument.

- The type parameter `RemainingItems` represents all the other arguments passed to the function

The notation `typename...` introduces a variadic template's **parameter pack**, which again represents **any number of arguments**. In the function's parameter list (line 9), the variable-length parameter must appear last and is denoted with `...` after the parameter's type (`RemainingItems`). Note that the `...` position is different in the template header and in the function parameter list. The return statement adds the `first` argument to the result of the **recursive call**

```
sum(theRest...)
```

The expression `theRest...` is a **parameter-pack expansion**—the compiler turns the parameter pack's elements into a comma-delimited list. We'll say more about this in a moment.

Calling `sum` with One Argument

When line 30 calls

```
sum(1)
```

which has one argument, the compiler invokes `sum`'s one-argument version (lines 8–11). If we have only one argument, the sum is simply the argument's value (in this case, 1).

Calling `sum` with Two Arguments

When line 30 calls

```
sum(1, 2)
```

the compiler invokes `sum`'s **variadic version** (lines 14–17), which receives 1 in the parameter `first` and 2 in the **parameter pack** `theRest`. The function then adds 1 and the result of calling `sum` with the **parameter-pack expansion** (`...`). The parameter pack contains only 2, so this call becomes `sum(2)`, which invokes `sum`'s one-argument version, ending the **recursion**. So, the final result is 3 (i.e., `1 + 2`).

Though `sum`'s **variadic version** looks like traditional **recursion**, the compiler generates separate template instantiations for each **recursive call**. So, the original `sum(1, 2)` call becomes

```
sum(1, sum(2))
```

Perf And because the compiler has all the values used in the calculation, it can perform the calculations and inline them in the program as compile-time constants, **eliminating execution-time function-call overhead**.⁷⁷

⁷⁷. “Template metaprogramming—Compile-time code optimization.” Accessed June 29, 2021. https://en.wikipedia.org/wiki/Template_metaprogramming#Compile-time_code_optimization.

Calling `sum` with Three Arguments

When line 31 calls

```
sum(1, 2, 3)
```

the compiler again invokes `sum`'s **variadic version** (lines 14–17):

- In this initial call, parameter `first` receives 1 and the **parameter pack** `theRest` receives 2 and 3. The function adds 1 and the result of calling `sum` with the **parameter-pack expansion**, producing the call `sum(2, 3)`.
- In the call `sum(2, 3)`, parameter `first` receives 2 and the **parameter pack** `theRest` receives 3. The function then adds 2 and the result of calling `sum` with the **parameter-pack expansion** producing the call `sum(3)`.
- The call `sum(3)` invokes `sum`'s one-argument version (the **base case**) with 3, which returns 3. At this point, the `sum(2, 3)` call's body becomes $2 + 3$ (that is 5), and the `sum(1, 2, 3)` call's body becomes $1 + 5$, producing the final result 6.

Effectively, the original call became

```
sum(1, sum(2, sum(3)))
```

where the compiler knows the value of the innermost call, `sum(3)`, and can determine the results of the other calls.

Calling `sum` with Three `string` Objects

Line 32's `sum` call

```
sum("s"s, "u"s, "m"s)
```

receives three `string`-object literals. Recall that `+` for `strings` performs **string concatenation**, so the original call effectively becomes

[Click here to view code image](#)

```
sum("s"s, sum("u"s, sum("m"s)))
```

producing the string "sum".

C++17 Fold Expressions

17 Fold expressions^{78,79,80} provide a convenient notation for repeatedly applying a binary operator to all the elements in a **variadic template's parameter pack**. Fold expressions are often used to **reduce the values in a parameter pack to a single value**. They also can **apply an operation to every object in a parameter pack**, such as calling a member function or displaying the pack's elements with `cout` (as we'll do in [Section 15.12.6](#)).

78. Jonathan Boccara, “C++ Fold Expressions 101,” March 12, 2021. Accessed June 17, 2021. <https://www.fluentcpp.com/2021/03/12/cpp-fold-expressions/>.

79. Jonathan Boccara, “What C++ Fold Expressions Can Bring to Your Code,” March 19, 2021. Accessed June 17, 2021. <https://www.fluentcpp.com/2021/03/19/what-c-fold-expressions-can-bring-to-your-code/>.

80. Jonathan Muller, “Nifty Fold Expression Tricks,” May 5, 2020. Accessed June 17, 2021. <https://www.foonathan.net/2020/05/fold-tricks/>.

Lines 20–23 use a **binary left fold** (line 22)

```
(first + ... + theRest)
```

which sums `first` and the zero or more arguments in the parameter pack `theRest`. **Fold expressions must be parenthesized. In a binary left fold, there are two binary operators, which must be the same**—in this case, the addition operator (`+`). The argument to the left of the first operator is the expression's **initial value**. The `...` expands the parameter pack to the right of the second operator, separating each parameter in the parameter pack from the next with the binary operator. So, in line 37, the function call

```
foldingSum(1, 2, 3)
```

the binary-left-fold expression expands to

(1 + 2 + 3)

If the parameter pack is empty, the value of the binary-left-fold expression is the initial value—in this case, the value of `first`. In Section 15.12.3, we'll discuss fold expressions in more detail.

 The C++ Core Guidelines recommend using **variadic function templates** for arguments of **mixed types**⁸¹ and using `initializer_lists` for functions that receive variable numbers of arguments of the **same type**.⁸² However, **fold expressions cannot be applied to `initializer_lists`**.

81. “T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types.” Accessed June 20, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic>.
82. “T.103: Don’t use variadic templates for homogeneous argument lists.” Accessed June 20, 2021. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic-not>.

15.12.3 Types of Fold Expressions

In the following descriptions,

- *pack* represents a parameter pack,
- *op* represents one of the 32 binary operators you can use in fold expressions,⁸³ and

83. “fold expression.” Accessed June 17, 2021. <https://en.cppreference.com/w/cpp/language/fold>.

- *initialValue* represents a starting value for a binary fold expression. For example, a sum might start with 0, or a product might start with 1.

There are **four fold-expression types**:

- A **unary left fold** has one binary operator with the parameter pack expansion `(...)` as the **left operand**, and the *pack* as the right operand:
$$(\dots \text{ } op \text{ } pack)$$
- A **unary right fold** has one binary operator with the parameter pack expansion `(...)` as the **right operand**, and the *pack* as the left operand:
$$(pack \text{ } op \text{ } \dots)$$

- A **binary left fold** has two binary operators, which must be the same. The *initial-Value* is to the **left of the first operator**, the parameter pack expansion (\dots) is between the operators, and the *pack* is to the right of the second operator:

$(initialValue \ op \ \dots \ op \ pack)$

- A **binary right fold** has two binary operators, which must be the same. The *initialValue* is to the **right of the second operator**, the parameter pack expansion (\dots) is between the operators, and *pack* is to the left of the first operator:

$(pack \ op \ \dots \ op \ initialValue)$

15.12.4 How Unary Fold Expressions Apply Their Operators

The key difference between left- and right-fold expressions is the order in which they apply their operators. **Left folds group left-to-right and right folds group right-to-left.** Depending on the operator, the grouping can produce different results. Figure 15.15 demonstrates unary left and right fold operations using the addition (+) and subtraction (-) operators:

- Lines 6–9 define **unaryLeftAdd**, which uses a **unary left fold** to add the items in its parameter pack.
- Lines 11–14 define **unaryRightAdd**, which uses a **unary right fold** operation to add the items in its parameter pack.
- Lines 16–19 define **unaryLeftSubtract**, which uses a **unary left fold** operation to subtract the items in its parameter pack.
- Lines 21–24 define **unaryRightSubtract**, which uses a **unary right fold** operation to subtract the items in its parameter pack.

[Click here to view code image](#)

```

1 // fig15_15.cpp
2 // Unary fold expressions.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 template <typename... Items>
```

```
7  auto unaryLeftAdd(Items... items) {
8      return (... + items); // unary left fold
9  }
10
11 template <typename... Items>
12 auto unaryRightAdd(Items... items) {
13     return (items + ...); // unary right fold
14 }
15
16 template <typename... Items>
17 auto unaryLeftSubtract(Items... items) {
18     return (... - items); // unary left fold
19 }
20
21 template <typename... Items>
22 auto unaryRightSubtract(Items... items) {
23     return (items - ...); // unary right fold
24 }
25
26 int main() {
27     std::cout << "Unary left and right fold "
28             << fmt::format("\n{}{}\n{}{}\n\n",
29                         "unaryLeftAdd(1, 2, 3, 4): ", u
30                         "unaryRightAdd(1, 2, 3, 4): ", u
31
32     std::cout << "Unary left and right fold "
33             << fmt::format("\n{}{}\n{}{}\n",
34                         "unaryLeftSubtract(1, 2, 3, 4): ",
35                         unaryLeftSubtract(1, 2, 3, 4),
36                         "unaryRightSubtract(1, 2, 3, 4) "
37                         unaryRightSubtract(1, 2, 3, 4))
38 }
```



Unary left and right fold with addition:
unaryLeftAdd(1, 2, 3, 4): 10
unaryRightAdd(1, 2, 3, 4): 10

```
Unary left and right fold with subtraction:  
unaryLeftSubtract(1, 2, 3, 4): -8  
unaryRightSubtract(1, 2, 3, 4): -2
```

Fig. 15.15 | Unary fold expressions.

Unary Left and Right Folds for Addition

Consider the line 29 call:

```
unaryLeftAdd(1, 2, 3, 4)
```

in which the **parameter pack** contains 1, 2, 3 and 4. This function performs a **unary left fold** using the + operator, which calculates:

$$(((1 + 2) + 3) + 4)$$

Similarly, the line 30 call:

```
unaryRightAdd(1, 2, 3, 4)
```

performs a **unary right fold**, which calculates:

$$(1 + (2 + (3 + 4)))$$

The order in which + is applied does not matter, so both expressions produce the same value (10) in this case.

Unary Left and Right Folds for Subtraction

The order in which some operators are applied can produce different results. Consider the call line 36 call:

[Click here to view code image](#)

```
unaryLeftSubtract(1, 2, 3, 4)
```

This function performs a **unary left fold** using the - operator, which calculates:

$$(((1 - 2) - 3) - 4)$$

producing -8. On the other hand, the line 37 call:

[Click here to view code image](#)

```
unaryRightSubtract(1, 2, 3, 4)
```

performs a **unary right fold** using the `-` operator, which calculates:

$$(1 - (2 - (3 - 4)))$$

producing `-2`.

Parameter Packs in Unary Fold Expressions Must Not Be Empty

Unary-fold expressions must be applied only to parameter packs with at least one element. The only exceptions to this are for the binary operators `&&`, `||` and comma `(,)`:

- An `&&` operation with an **empty parameter pack** evaluates to `true`.
- An `||` operation with an **empty parameter pack** evaluates to `false`.
- Any operation performed with the **comma operator** on an **empty parameter pack** evaluates to `void()`, which simply means no operation is performed. We show a fold expression using the comma operator in [Section 15.12.6](#).

15.12.5 How Binary-Fold Expressions Apply Their Operators

If an **empty parameter pack** is possible, you can use a **binary left fold** or **binary right fold**. Each requires an initial value. If the **parameter pack** is empty, the initial value is used as the fold expression's value. [Figure 15.16](#) demonstrates **binary left folds** and **binary right folds** for addition and subtraction:

- Lines 6–9 define `binaryLeftAdd`, which uses a **binary left fold** to add the items in its parameter pack starting with the initial value 0.
- Lines 11–14 define `binaryRightAdd`, which uses a **binary right fold** to add the items in its parameter pack starting with the initial value 0.
- Lines 16–19 define `binaryLeftSubtract`, which uses a **binary left fold** to subtract the items in its parameter pack starting with the initial value 0.
- Lines 21–24 define `binaryRightSubtract`, which uses a **binary right fold** to subtract the items in its parameter pack starting with the

initial value 0.

Once again, note that the fold expressions' grouping for addition does not matter, but for subtraction leads to different results.

[Click here to view code image](#)

```
1 // fig15_16.cpp
2 // Binary fold expressions.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 template <typename... Items>
7 auto binaryLeftAdd(Items... items) {
8     return (0 + ... + items); // binary left
9 }
10
11 template <typename... Items>
12 auto binaryRightAdd(Items... items) {
13     return (items + ... + 0); // binary right
14 }
15
16 template <typename... Items>
17 auto binaryLeftSubtract(Items... items) {
18     return (0 - ... - items); // binary left
19 }
20
21 template <typename... Items>
22 auto binaryRightSubtract(Items... items) {
23     return (items - ... - 0); // binary right
24 }
25
26 int main() {
27     std::cout << "Binary left and right fold
28             << fmt::format("\n{}{}\n{}{}\n{}{}\n{
29                 "binaryLeftAdd(): ", binaryLeftAdd(),
30                 "binaryLeftAdd(1, 2, 3, 4): ",
```

```

31         "binaryRightAdd(): ", binaryRightAdd(),
32         "binaryRightAdd(1, 2, 3, 4): ",
33
34     std::cout << "Binary left and right fold"
35             << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}")
36             "binaryLeftSubtract(): ", binaryLeftSubtract(),
37             "binaryLeftSubtract(1, 2, 3, 4)"
38             binaryLeftSubtract(1, 2, 3, 4),
39             "binaryRightSubtract(): ", binaryRightSubtract(),
40             "binaryRightSubtract(1, 2, 3, 4)"
41             binaryRightSubtract(1, 2, 3, 4)
42 }
```

< **>**

Binary left and right fold with addition:

```

binaryLeftAdd(): 0
binaryLeftAdd(1, 2, 3, 4): 10
binaryRightAdd(): 0
binaryRightAdd(1, 2, 3, 4): 10
```

Binary left and right fold with subtraction:

```

binaryLeftSubtract(): 0
binaryLeftSubtract(1, 2, 3, 4): -10
binaryRightSubtract(): 0
binaryRightSubtract(1, 2, 3, 4): -2
```

Fig. 15.16 | Binary fold expressions.

15.12.6 Using the Comma Operator to Repeatedly Perform an Operation

The **comma operator** evaluates the expression to its left, then the expression to its right. The value of a comma-operator expression is the value of the rightmost expression. You can **combine the comma operator with fold expressions to repeatedly perform tasks for every item in a parameter pack**. Figure 15.17's `printItems` function displays every item in its parameter pack (`items`) on a line by itself. Line 9 executes the expression

on the left of the comma operator:

[Click here to view code image](#)

```
(std::cout << items << "\n")
```

once for each item in the parameter pack `items`.

[Click here to view code image](#)

```
1 // fig15_17.cpp
2 // Repeating a task using the comma operator
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <vector>
6
7 template <typename... Items>
8 void printItems(Items... items) {
9     ((std::cout << items << "\n"), ...); // :
10 }
11
12 int main() {
13     std::cout << "printItems(1, 2.2, \"hello"
14     printItems(1, 2.2, "hello");
15 }
```

< >

```
printItems(1, 2.2, "hello"):
1
2.2
hello
```

Fig. 15.17 | Repeating a task using the comma operator and fold expressions.

15.12.7 Constraining Parameter Pack Elements to the Same Type

When using **fold expressions**, there may be cases in which you'd like every element to have the **same type**. For example, you might want a variadic function template to sum its arguments and produce a result of the same type as the arguments. Figure 15.18 uses the **predefined concept std::same_as** to check that all the elements of a parameter pack have the same type as another type argument.

[Click here to view code image](#)

```
1 // fig15_18.cpp
2 // Constraining a variadic-function-template
3 // elements of the same type.
4 #include <concepts>
5 #include <iostream>
6 #include <string>
7
8 template <typename T, typename... U>
9 concept SameTypeElements = (std::same_as<T,
10
11 // add one or more arguments with a fold expression
12 template <typename FirstItem, typename... RemainingI
13     requires SameTypeElements<FirstItem, RemainingI>
14     auto foldingSum(FirstItem first, RemainingI... theRest)
15         return (first + ... + theRest); // expands to
16 }
17
18 int main() {
19     using namespace std::literals;
20
21     foldingSum(1, 2, 3); // valid: all are integers
22     foldingSum("s"s, "u"s, "m"s); // valid: all are strings
23     foldingSum(1, 2.2, "hello"s); // error: not all are the same type
24 }
```

< >

```
fig15_18.cpp:23:4: error: no matching function for call to 'foldingSum'
```

```
foldingSum(1, 2.2, "hello"s); // error: three
^~~~~~  
  
fig15_18.cpp:14:6: note: candidate template ignored
[with FirstItem = int, RemainingItems = <double,
auto foldingSum(FirstItem first, RemainingItems.
    ^
  
  
fig15_18.cpp:13:13: note: because sic_string<char>
requires SameTypeElements<FirstItem, RemainingItems>
    ^
  
  
fig15_18.cpp:9:34: note: because false
concept SameTypeElements = (std::same_as<T, U> &&
    ^
  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../...
concepts:63:19: note: because false
= __detail::__same_as<_Tp, _Up> && __detail::__s
    ^
  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../...
concepts:57:27: note: because false
    concept __same_as = std::is_same_v<_Tp, _U>
        ^
  
  
fig15_18.cpp:9:34: note: and evaluated to false
concept SameTypeElements = (std::same_as<T, U> &&
    ^
  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../...
concepts:63:19: note: because sic_string<char> >
    = __detail::__same_as<_Tp, _Up> && __detail::__s
        ^
  
  
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/.../...
concepts:57:27: note: because evaluated to false
```

```
concept __same_as = std::is_same_v<Tp, U>
^

1 error generated.
```

Fig. 15.18 | Constraining a variadic-function-template parameter pack to elements of the same type. (Part 2 of 2.)

Custom Concept for a Variadic Template

Concepts Lines 8–9 define a custom concept `SameTypeElements` with two type parameters:

- the first represents a single type, and
- the second is a **variadic type parameter**.

The constraint

```
std::same_as<T, U>
```

compares the type `T` to one type from the parameter pack `U`. This constraint is in a fold expression, so the **parameter pack expansion** (`...`) applies `std::same_as<T, U>` once for each type from the parameter pack `U`, comparing every type in `U` to the type `T`. If every `U` is the same as `T`, then the constraint evaluates to `true`. Line 13 in the `foldingSum` function template applies our `SameTypeElements` concept to the function's type parameters.

Calling `foldingSum`

Err In `main`, lines 21 and 22 call `foldingSum` with three `ints` and three `strings`, respectively. Each call has three arguments of the same type, so these statements will successfully compile. However, line 23 attempts to call `foldingSum` with an `int`, a `double` and a `string`. In that call, type `T` in the `SameTypeElements` concept is `int`, and types `double` and `string` are placed in the parameter pack `U`. Of course, both `double` and `string` are not `int`, so each use of the concept `std::same_as<T, U>` fails. [Figure 15.18](#) shows the **Clang** compiler error messages. We highlighted

key messages in bold and added vertical spacing for readability. Clang indicates:

[Click here to view code image](#)

```
error: no matching function for call to
```

and

[Click here to view code image](#)

```
note: candidate template ignored: constraints not
```



It also points out in its notes each individual `std::same_as` test that failed because a type was not the same as `int`:

```
note: because
```

```
note: and false
```

15.13 Template Metaprogramming

As we've mentioned, a goal of modern C++ is to do work at compile time to enable the compiler to optimize your code for runtime performance. Much of this optimization work is done via **template metaprogramming (TMP)**, which enables the compiler to

- manipulate types,
- perform compile-time computations, and
- generate optimized code.

The **concepts**, **concept-based overloading** and **type traits** capabilities we've introduced in this chapter all are crucial template-metaprogramming capabilities.

 **SE** Template metaprogramming is complex. When properly used, it can help you improve the runtime performance of your programs. So it's important to be aware of template metaprogramming's powerful capabilities.

Our goal in this section is to present a variety of manageable template-metaprogramming examples and to provide in our footnote citations resources for further study. We'll show examples demonstrating:

- computing values at compile-time with metafunctions,
- computing values at compile-time with `constexpr` functions,
- using type traits at compile-time via the `constexpr if` statement to optimize runtime performance, and
- manipulating types at compile-time with metafunctions.

15.13.1 C++ Templates Are Turing Complete

Todd Veldhuizen proved that C++ templates are **Turing complete**,^{84,85} so anything that can be computed at runtime with traditional C++, also can be computed at compile-time with C++ template metaprogramming. One of the first attempts to demonstrate such compile-time computation was presented in 1994 during C++'s early standardization efforts. Erwin Unruh wrote a template metaprogram to calculate the prime numbers less than 30.^{86,87} Though the program did not compile, the compiler's error messages included the prime-number calculation results. You can view the original program—which is no longer valid C++—and the original compiler error messages at:

84. Todd Veldhuizen, “C++ Templates are Turing Complete,” 2003. Accessed July 2, 2021. <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670>.
85. “Template metaprogramming.” Accessed July 2, 2021. https://en.wikipedia.org/wiki/Template_metaprogramming.
86. Erwin Unruh, “Prime Number Computation,” 1994. ANSI X3J16-94-0075/ISO WG21-4-62.
87. Rainer Grimm, “C++ Core Guidelines: Rules for Template Metaprogramming,” January 7, 2019. Accessed July 2, 2021. <https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-template-metaprogramming>. Our thanks to Rainer Grimm for this blog post that pointed us to the historical note about Erwin Unruh.

[Click here to view code image](#)

<http://www.erwin-unruh.de/primorig.html>

Here are a few of those error messages. The specializations of his class template D showing the first several prime numbers are highlighted in bold:

[Click here to view code image](#)

```
| Type D<2>' ("primes.cpp", L2/C25) .
| Type D<3>' ("primes.cpp", L2/C25) .
```

```
|     Type D<5>' ("primes.cpp", L2/C25) .
|     Type D<7>' ("primes.cpp", L2/C25) .
```

15.13.2 Computing Values at Compile Time

Perf  The goal of compile-time calculations is to optimize a program's runtime performance.⁸⁸ Figure 15.19 uses template metaprogramming to calculate factorials at compile time. First, we use a recursive factorial definition implemented with templates. Then, we show two compile-time `constexpr` functions using the iterative and recursive algorithms presented in Section 5.18. The C++ Core Guidelines recommend using `constexpr` functions to compute values at compile time⁸⁹—such functions use traditional C++ syntax.

88. “Template metaprogramming.” Accessed June 24, 2021. https://en.wikipedia.org/wiki/Template_metaprogramming.

[Click here to view code image](#)

```
1 // fig15_19.cpp
2 // Calculating factorials at compile time.
3 #include <iostream>
4
5 // Factorial value metafunction calculates
6 template <int N>
7 struct Factorial {
8     static constexpr long long value{N * Fac
9 };
10
11 // Factorial specialization for the base ca
12 template <>
13 struct Factorial<0> {
14     static constexpr long long value{1}; // 
15 };
16
17 // constexpr compile-time recursive factori
18 constexpr long long recursiveFactorial(int :
```

```
19     if (number <= 1) {
20         return 1; // base cases: 0! = 1 and 1
21     }
22     else { // recursion step
23         return number * recursiveFactorial(number - 1);
24     }
25 }
26
27 // constexpr compile-time iterative factorial
28 constexpr long long iterativeFactorial(int number) {
29     long long factorial{1}; // result for 0!
30
31     for (long long i{2}; i <= number; ++i) {
32         factorial *= i;
33     }
34
35     return factorial;
36 }
37
38 int main() {
39     // "calling" a value metaprogram requires
40     // the template and accessing its static
41     std::cout << "CALCULATING FACTORIALS AT "
42             << "\nWITH A RECURSIVE METAPROGRAM"
43             << "\nFactorial(0): " << Factorial<0>()
44             << "\nFactorial(1): " << Factorial<1>()
45             << "\nFactorial(2): " << Factorial<2>()
46             << "\nFactorial(3): " << Factorial<3>()
47             << "\nFactorial(4): " << Factorial<4>()
48             << "\nFactorial(5): " << Factorial<5>()
49
50     // calling the recursive constexpr function
51     std::cout << "\n\nCALCULATING FACTORIALS
52             WITH A RECURSIVE CONSTEXPR FUNC"
53             << "\nrecursiveFactorial(0): " << recursiveFactorial(0)
54             << "\nrecursiveFactorial(1): " << recursiveFactorial(1)
55             << "\nrecursiveFactorial(2): " << recursiveFactorial(2)
```

```
56      << "\nrecursiveFactorial(3): " << rec
57      << "\nrecursiveFactorial(4): " << rec
58      << "\nrecursiveFactorial(5): " << rec
59
60 // calling the iterative constexpr function
61 std::cout << "\n\nCALCULATING FACTORIALS
62             << "\nWITH AN ITERATIVE CONSTEXPR FUNCTION"
63             << "\nniterativeFactorial(0): " << ite
64             << "\nniterativeFactorial(1): " << ite
65             << "\nniterativeFactorial(2): " << ite
66             << "\nniterativeFactorial(3): " << ite
67             << "\nniterativeFactorial(4): " << ite
68             << "\nniterativeFactorial(5): " << ite
69 }
```

< >

CALCULATING FACTORIALS AT COMPILE-TIME WITH A RECURSIVE METAFUNCTION

```
Factorial(0): 1
Factorial(1): 1
Factorial(2): 2
Factorial(3): 6
Factorial(4): 24
Factorial(5): 120
```

CALCULATING FACTORIALS AT COMPILE-TIME WITH A RECURSIVE CONSTEXPR FUNCTION

```
recursiveFactorial(0): 1
recursiveFactorial(1): 1
recursiveFactorial(2): 2
recursiveFactorial(3): 6
recursiveFactorial(4): 24
recursiveFactorial(5): 120
```

CALCULATING FACTORIALS AT COMPILE-TIME WITH AN ITERATIVE CONSTEXPR FUNCTION

```
iterativeFactorial(0): 1
```

```
iterativeFactorial(1) : 1
iterativeFactorial(2) : 2
iterativeFactorial(3) : 6
iterativeFactorial(4) : 24
iterativeFactorial(5) : 120
```

Fig. 15.19 | Calculating factorials at compile time. (Part 3 of 3.)

Metafunctions

You can perform compile-time calculations with **metafunctions**. Like the functions you've defined so far, metafunctions have arguments and return values, but their syntax is significantly different. **Metafunctions are implemented as class templates**—typically using structs. A metafunction's arguments are the items used to specialize the class template, and its return value is a public class member. The type traits introduced in [Section 15.5](#) are metafunctions.

There are two types of metafunctions:

- A **value metafunction** is a class template with a `public static constexpr` **data member named `value`**. The class template uses its template arguments to compute `value` at compile time. This is how we implement factorial calculations.
- A **type metafunction** is a class template with a **nested `type` member**, typically defined as a **type alias**. The class template uses its template arguments to manipulate a type at compile time. [Section 15.13.4](#) presents type metafunctions.

The metafunction member names `value` and `type` are conventions⁹⁰ used throughout the C++ standard library and in metaprogramming in general.

90. Jody Hagins, “Template Metaprogramming: Type Traits (part 1 of 2),” YouTube video, September 22, 2020. Accessed June 15, 2021. <https://www.youtube.com/watch?v=tiAVWcjIF6o>.

Factorial Metafunction

Our Factorial metafunction (lines 6–9) is implemented using the recursive factorial definition:

$$n! = n \cdot (n - 1)!$$

Factorial has one **non-type template parameter**—the int parameter N (line 6)—representing the metafunction’s argument. Per the C++ standard’s conventions for value meta-functions, line 8 defines a public static `constexpr` data member named `value`. Factorials grow quickly, so we declared the variable’s type as `long long`. The constant value is initialized with the result of the expression:

```
N * Factorial<N - 1>::value
```

which multiplies N by the result of “calling” the Factorial metafunction for $N - 1$.

“Calling” a Metafunction

You “call” a metafunction by specializing its template, which probably feels weird to you. For example, to calculate the factorial of 3, you’d use the specialization:

```
Factorial<3>::value
```

in which 3 is the **template argument** and `::value` is the **return value**. This expression causes the compiler to create a new type representing the factorial of 3. We’ll say more about this momentarily.

Factorial Metafunction Specialization for the Base Case

—Factorial<0>

For the recursive factorial calculation, $0!$ is the **base case**, which is defined to be 1. In **meta-function recursion**, you specify the base case with a **full template specialization** (lines 12–15). Such a specialization uses the notation `template <>` (line 12) to indicate that all the template’s arguments will be specified explicitly in the angle brackets following the class name. The full template specialization `Factorial<0>` matches only Factorial calls with the argument 0. Line 14 sets the specialization’s `value` member to be 1.⁹¹

91. Section 5.18 specified that 0 and 1 are both base cases for factorial calculations. To mimic that, we could implement a second full template specialization named `Factorial<1>`. As implemented, the Factorial metafunction handles `Factorial<1>` as `1 * Factorial<0>`.

How the Compiler Evaluates Metafunction Factorial for the

Argument 0

When the compiler encounters a metafunction call, such as `Factorial<0>::value` (line 41 in `main`), it must determine which `Factorial` class template to use. The template argument is the `int` value 0. The class template `Factorial` in lines 6–9 can match any `int` value, and the full-template specialization `Factorial<0>` in lines 12–15 specifically matches only the value 0. **The compiler always chooses the most specialized template from multiple matching templates.** So, `Factorial<0>` matches the full template specialization, and `Factorial<0>::value` evaluates to 1.

How the Compiler Evaluates Metafunction `Factorial` for the Argument 3

Now, let's reconsider the metafunction call `Factorial<3>::value` (line 46 in `main`). The compiler generates each specialization needed to produce the final result at compile time. When the compiler encounters `Factorial<3>::value`, it specializes the class template in lines 6–9 with 3 as the argument. In doing so, line 8 of the specialization becomes

```
3 * Factorial<2>::value
```

The compiler sees that to complete the `Factorial<3>` definition, it must specialize the template again for `Factorial<2>`. In that specialization, line 8 becomes

```
2 * Factorial<1>::value
```

Next, the compiler sees that to complete the `Factorial<2>` definition, it must specialize the template again for `Factorial<1>`. In that specialization, line 8 becomes

```
1 * Factorial<0>::value
```

The compiler knows that `Factorial<0>::value` is 1 from the full specialization in lines 12–15. This completes the `Factorial` specializations for `Factorial<3>`.

Now, the compiler knows everything it needs to complete the earlier `Factorial` specializations:

- `Factorial<1>::value` stores the result of $1 * 1$, which is 1.
- `Factorial<2>::value` stores the result of $2 * 1$, which is 2.
- `Factorial<3>::value` stores the result of $3 * 2$, which is 6.

Perf  So the compiler can insert the final constant value 6 in place of the original metafunction call—**without any runtime overhead**. Each of the Factorial metafunction calls in lines 43–48 of main are evaluated similarly by the compiler, **enabling it to optimize the program's performance by removing computations from runtime**.

Functional Programming

None of what the compiler does during **template specialization** modifies variable values after they're initialized.⁹² **There are no mutable variables in template metaprograms.** This is a hallmark of **functional programming**. All the values processed in this example are compile-time constants.

92. “Template metaprogramming.” Accessed June 24, 2021.
https://en.wikipedia.org/wiki/Template_metaprogramming.

Using `constexpr` Functions to Perform Compile-Time Calculations

As you can see, using metafunctions to calculate values at compile time is not as straightforward as using traditional runtime functions. C++ provides two options for compile-time evaluation of traditional functions:

- A function declared `constexpr` can be **called at compile-time or runtime**, but it must produce a constant.
- **SE**  In C++20, you can declare a function `consteval` (rather than `constexpr`) to indicate that it can be **called only at compile-time** to produce a constant.

SE  Many computations performed with metafunctions are easier to implement using traditional functions that are declared `constexpr` or `consteval`. Such functions also are part of compile-time metaprogramming. In fact, various members of the C++ standard committee prefer `constexpr`-based metaprogramming over template-based metaprogramming, which is “difficult to use, does not scale well and is basically equivalent to inventing a new language within C++.”⁹³ They've

proposed various `constexpr` enhancements to simplify other aspects of metaprogramming in future C++ versions.

[93.](#) Louis Dionne, “Metaprogramming by design, not by accident,” June 18, 2017. Accessed June 30, 2021. <https://wg21.link/p0425r0>.

Lines 18–25 and 28–36 define the `recursiveFactorial` and `iterativeFactorial` functions using the algorithms from [Section 5.18](#) and traditional C++ syntax. Each function is declared `constexpr`, so the compiler can evaluate the function’s result at compile time. Lines 53–58 demonstrate `recursiveFactorial`, and lines 63–68 demonstrate `iterativeFactorial`.

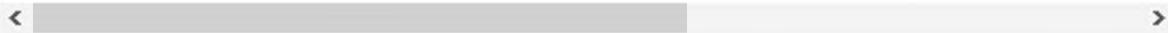
15.13.3 Conditional Compilation with Template Metaprogramming and `constexpr if`

[20](#) [17](#) Another aspect of doing more at compile time to optimize runtime execution is generating code that executes optimally at runtime. [Section 15.6.4](#) showed the C++20 way to optimize runtime execution for a `customDistance` function template using **concept-based function overloading**. In that example, the compiler chose the correct overloaded function template to call based on the template parameter’s type constraints. Here, we’ll reimplement that example using a single `customDistance` function template. Inside the function, a C++17 **compile-time if statement**—known as a `constexpr if`—uses the `std::is_same` type trait (introduced in [Section 15.12.7](#)) to decide whether to generate the $O(1)$ or $O(n)$ distance calculation based on the function’s iterator arguments.

[Click here to view code image](#)

```
1 // fig15_20.cpp
2 // Implementing customDistance using template
3 #include <array>
4 #include <iostream>
5 #include <iterator>
6 #include <list>
7 #include <ranges>
8 #include <type_traits>
```

```
9
10 // calculate the distance (number of items)
11 // requires at least input iterators
12 template <typename Iterator>
13     requires std::input_iterator<Iterator>
14 auto customDistance(Iterator begin, Iterator end) {
15     // for random-access iterators subtract
16     if constexpr (std::is_same<Iterator::iterator_category,
17                     std::random_access_iterator_tag>::value)
18
19         std::cout << "customDistance with ran-
20         return end - begin; // O(1) operation
21     }
22     else { // for all other iterators
23         std::cout << "customDistance with non-
24         std::ptrdiff_t count{0};
25
26         // increment from begin to end and co-
27         // O(n) operation for non-random-acce
28         for (auto& iter{begin}; iter != end;
29             ++count);
30     }
31
32     return count;
33 }
34 }
35
36 int main() {
37     const std::array ints1{1, 2, 3, 4, 5}; /
38     const std::list ints2{1, 2, 3}; // has b
39
40     auto result1{customDistance(ints1.begin(
41         std::cout << "ints1 number of elements:
42     auto result2{customDistance(ints2.begin(
43         std::cout << "ints1 number of elements:
44 }
```



```
customDistance with random-access iterators  
ints1 number of elements: 5  
customDistance with non-random-access iterators  
ints1 number of elements: 3
```

Fig. 15.20 | Implementing `customDistance` using template metaprogramming. (Part 2 of 2.)

Lines 12–34 define function template `customDistance`. Since this function requires **at least input iterators** to perform its task, line 13 constrains the type parameter `Iterator` using the **concept std::input_iterator**. Though the **compile-time if statement** is known in the C++ standard as a **constexpr if**, it's written in code as **if constexpr** (line 16). This statement's condition must evaluate at compile-time to a `bool`.

Lines 16–17 use the **std::is_same type trait** to compare two types:

- `Iterator::iterator_category` and
- `std::random_access_iterator_tag`.

Recall that a standard iterator has a **nested type named iterator_category**, which designates the iterator's type using one of the following "tag" types from header `<iterator>`:

- `input_iterator_tag`,
- `output_iterator_tag`,
- `forward_iterator_tag`,
- `bidirectional_iterator_tag`,
- `random_access_iterator_tag`, or
- `contiguous_iterator_tag`.

In line 16, the expression `Iterator::iterator_category` gets the iterator tag from the argument's iterator type, which the **metafunction std::is_same** then compares to

[Click here to view code image](#)

```
std::random_access_iterator_tag
```

The comparison result is true or false, which we access via **metafunction `std::is_same`'s `value` member**. If `value` is true, the compiler instantiates the code in the `if constexpr` body (lines 19–20), which is optimized for **random-access iterators** to perform the **$O(1)$** calculation. Otherwise, the compiler instantiates `else`'s body (lines 23–32), which uses the **$O(n)$** approach that works for all other iterator types.

15.13.4 Type Metafunctions

SE  **Type metafunctions frequently are used to add attributes to and remove attributes from types at compile-time.** This is a more advanced template-metaprogramming technique used primarily by template class-library implementers. Here, we'll add and remove type attributes using our own type metafunctions that mimic ones from the **`<type_traits>` header** so you can see how they might be implemented. We'll also use predefined ones. **Always prefer the `<type_traits>` header's predefined type traits rather than implementing your own.**

[Click here to view code image](#)

```
1 // fig15_21.cpp
2 // Adding and removing type attributes with
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <type_traits>
6
7 // add const to a type T
8 template <typename T>
9 struct my_add_const {
10     using type = const T;
11 };
12
13 // general case: no pointer in type, so set
14 template <typename T>
15 struct my_remove_ptr {
```

```
16     using type = T;
17 }
18
19 // partial template specialization: T is a ]
20 template <typename T>
21 struct my_remove_ptr<T*> {
22     using type = T;
23 }
24
25 int main() {
26     std::cout << fmt::format("{}\n{}{}\n{}{}")
27         "ADD CONST TO A TYPE VIA A CUSTOM TYP"
28         "std::is_same<const int, my_add_const>"
29         "std::is_same<const int, my_add_const<"
30         "std::is_same<int* const, my_add_cons"
31         "std::is_same<int* const, my_add_const"
32
33     std::cout << fmt::format("{}\n{}{}\n{}{}")
34         "REMOVE POINTER FROM TYPES VIA A CUST"
35         "std::is_same<int, my_remove_ptr<int>>"
36         "std::is_same<int, my_remove_ptr<int>::"
37         "std::is_same<int, my_remove_ptr<int*>::"
38         "std::is_same<int, my_remove_ptr<int*>::"
39
40     std::cout << fmt::format("{}\n{}{}\n{}{}")
41         "ADD REFERENCES TO TYPES USING STANDA"
42         "std::is_same<int&, std::add_lvalue_re"
43         "std::is_same<int&, std::add_lvalue_re"
44         "std::is_same<int&&, std::add_rvalue_re"
45         "std::is_same<int&&, std::add_rvalue_re"
46
47     std::cout << fmt::format("{}\n{}{}\n{}{}")
48         "REMOVE REFERENCES FROM TYPES USING S"
49         "std::is_same<int, std::remove_refere"
50         "std::is_same<int, std::remove_referen"
51         "std::is_same<int, std::remove_refere"
52         "std::is_same<int, std::remove_referen
```

```
53     "std::is_same<int, std::remove_reference<
54         std::is_same<int, std::remove_reference<
55     }
```

```
ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION
std::is_same<const int, my_add_const<int>::type>
std::is_same<int* const, my_add_const<int*>::type>
```

```
REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION
std::is_same<int, my_remove_ptr<int>::type>::value
std::is_same<int, my_remove_ptr<int*>::type>::value
```

```
ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS
std::is_same<int&, std::add_lvalue_reference<int>::type>
std::is_same<int&&, std::add_rvalue_reference<int>::type>
```

```
REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS
std::is_same<int, std::remove_reference<int>::type>
std::is_same<int, std::remove_reference<int&>::type>
std::is_same<int, std::remove_reference<int&&>::type>
```

Fig. 15.21 | Compile-time type manipulation. (Part 2 of 2.)

Adding `const` to a Type

Lines 8–11 implement a **type metaprogramming** named `my_add_const` that's a simplified version of the `std::add_const` metaprogram from header `<type_traits>`. By convention, a **type metaprogramming must have a public member named `type`**. When `my_add_const` is specialized with a type, the compiler defines the type alias in line 10, which precedes the type with `const`. The **predefined `std::add_const` metaprogram** is more elaborate. It also checks whether the type `T` is a reference, a function, or a `const`-qualified type and, if so, sets its `type` member to that same type; otherwise, it applies `const` to the type.⁹⁴

94. “20.15.8.2 Const-volatile modifications [meta.trans.cv].” Accessed June 26, 2021.

<https://eel.is/c++draft/meta#trans.cv>.

Lines 26–31 in `main` demonstrate the **`my_add_const` metaprogramming**. Line 29

```
my_add_const<int>::type
```

specializes the template with the non-const type `int`. To confirm that `const` was added to `int`, we use the **`type trait std::is_same`** to compare the type `const int` to the type returned by the preceding expression. They are the same, so `std::is_same`'s `value` member is `true`, as shown in the output.

Removing * from a Pointer Type

Lines 14–17 and 20–23 implement a **custom type metaprogramming** named `my_remove_ptr` that mimics the `std::remove_pointer` metaprogramming from the `<type_traits>` header to show how that metaprogramming could be implemented. This requires two metaprogramming class templates:

- The one in lines 14–17 handles the general case in which a type is not a pointer. Line 16's type alias defines the `type` member simply as `T`, which can be specialized with any type. For any type that does not include an `*` to indicate a pointer, this metaprogramming simply returns the type used to specialize the template.
- The metaprogramming class template in lines 20–23 matches only pointer types. For this purpose, we define a **partial template specialization**.⁹⁵ Unlike a **full template specialization** that begins with the **template header `template<>`**, a partial template specialization still specifies a template header (line 20) with one or more parameters, which it uses in the angle brackets following the class name (line 21). In this case, the partial specialization is that `T` must be a pointer—indicated with `T*`. To remove the `*` from the pointer type, the type alias in line 22 defines the `type` member simply as `T`.

⁹⁵ Inbal Levi. “Exploration of C++20 Metaprogramming.” September 29, 2020. Accessed June 14, 2021. <https://www.youtube.com/watch?v=XgrjybKaIV8>.

Lines 36 and 38 in `main` demonstrate our **custom `my_remove_ptr` type metaprogramming**. Let's consider the compiler's matching process for invoking

them.

Specializing `my_remove_pointer` for Non-Pointer Types

The expression in line 36

```
my_remove_ptr<int>::type
```

specializes the template with the non-pointer type `int`. The compiler must decide which definition of `my_remove_ptr` matches the template specialization. Since there is no `*` to indicate a pointer in the type `int`, this specialization matches only the `my_remove_ptr` definition in lines 14–17, which sets its `type` member to the type argument `int`. To confirm this, we use the `type trait std::is_same` to compare type `int` to the type returned by the preceding expression. They are the same, so `std::is_same`'s `value` member is `true`.

Specializing `my_remove_pointer` for Pointer Types

The expression in line 38

```
my_remove_ptr<int*>::type
```

specializes the template with the pointer type `int*`, which can match both definitions of `my_remove_const`:

- the first definition can match any type, and
- the second can match any pointer type.

Again, the **compiler always chooses the most specific matching template**. In this case, the type `int*` matches the **partial template specialization** in lines 20–23:

- the `int` in the preceding expression matches `T` in line 21, and
- the `*` in the preceding expression matches the `*` in line 21, separating it from the type `int`—this is what enables the **partial template specialization** to remove the pointer (`*`) from the type.

To confirm that the `*` was removed, we use `std::is_same` to compare `int` to the type returned by the preceding expression. They are the same, so `std::is_same`'s `value` member is `true`.

Adding *lvalue* and *rvalue* References to a Type

The **predefined type metafunctions** that modify types work similarly to our `my_add_const` and `my_remove_ptr` type metafunctions. Lines 43 and 45 in `main` demonstrate the **predefined type metafunctions `add_lvalue_reference` and `add_rvalue_reference`**. Line 43 converts type `int` to type `int&`—an ***lvalue reference type***. Line 45 converts type `int` to type `int&&`—an ***rvalue reference type***. We use `std::is_same` to confirm both results.

Removing *lvalue* and *rvalue* References from a Reference Type

Lines 50, 52 and 54 test the `<type_traits>` header’s **predefined `std::remove_reference` metafunction**, which removes *lvalue* or *rvalue* references from a type. We specialize `std::remove_reference` with the types `int`, `int&` and `int&&`, respectively:

- non-reference types (like `int`) are returned as is,
- *lvalue* reference types have the `&` removed, and
- *rvalue* reference types have the `&&` removed.

To confirm this, we use `std::is_same` to compare `int` to the type returned by each expression in lines 50, 52 and 54. They are the same, so `std::is_same`’s `value` member is `true` in each case.

15.14 Wrap-Up

In this chapter, we demonstrated the power of generic programming and compile-time polymorphism with templates and concepts. We used class templates to create related custom classes, distinguishing between the templates you write and template specializations the compiler generates from your code.

We introduced C++20’s abbreviated function templates and templated lambdas. We used C++20 concepts to constrain template parameters and overload function templates based on their type requirements. Next, we introduced type traits and showed how they relate to C++20 concepts. We created our own custom concept and showed how to test it at compile-time with `static_assert`.

We demonstrated how to create a custom concept-constrained algorithm, then used it to manipulate objects of several C++ standard library container

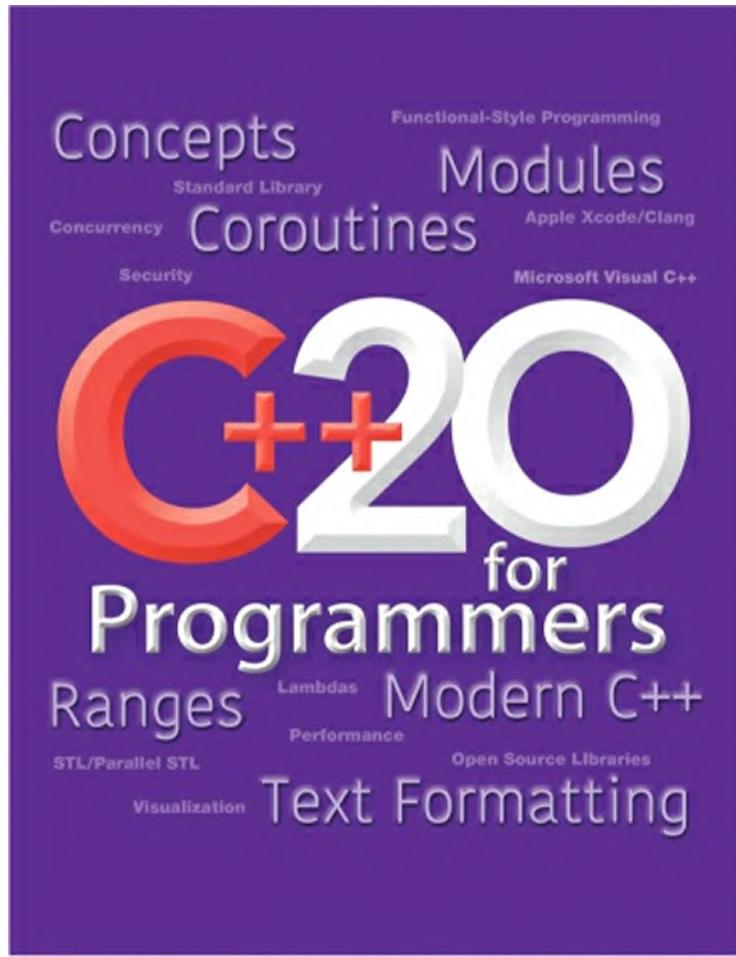
classes. Next, we rebuilt our class `MyArray` as a custom container class template with custom iterators. We showed that those iterators enabled C++ standard library algorithms to manipulate `MyArray` elements. We also introduced non-type template parameters for passing compile-time constants to templates and discussed default template arguments.

We demonstrated that variadic templates can receive any number of parameters, first using the standard library's `tuple` variadic class template, then with variadic function templates. We also showed how to apply binary operators to variadic parameter packs using fold expressions.

Many of the techniques we demonstrated are newer ways to perform various aspects of compile-time metaprogramming. We concluded with an introduction to several other metaprogramming techniques, including how to compute values at compile time with value metafunctions, how to perform compile-time conditional compilation with `constexpr if` and how to modify types at compile time with type metafunctions.

In the next chapter, we present C++ techniques that enable you to take advantage of your system's multicore architecture—concurrency, parallelism, C++20 coroutines and the parallel STL algorithms.

Chapter 16. C++20 Modules: Large-Scale Development



Objectives

In this chapter, you'll:

- Understand the motivation for modularity, especially for large software systems.
- See how modules improve encapsulation.
- import standard library headers as module header units.
- Define a module's primary interface unit.

- export declarations from a module to make them available to other translation units.
 - import modules to use their exported declarations.
 - Separate a module's interface from its implementation by placing the implementation in a `:private` module fragment or a module implementation unit.
 - See what compilation errors occur when you attempt to use non-exported module items.
 - Use module partitions to organize modules into logical components.
 - Divide a module into “submodules” so client-code developers can choose the portion(s) of a library they wish to use.
 - Understand visibility vs. reachability of declarations.
 - See how modules can reduce translation unit sizes and compilation times.
-

Outline

[16.1 Introduction](#)

[16.2 Compilation and Linking Prior to C++20](#)

[16.3 Advantages and Goals of Modules](#)

[16.4 Example: Transitioning to Modules— Header Units](#)

[16.5 Example: Creating and Using a Module](#)

[16.5.1 module Declaration for a Module Interface Unit](#)

[16.5.2 Exporting a Declaration](#)

[16.5.3 Exporting a Block of Declarations](#)

[16.5.4 Exporting a namespace](#)

[16.5.5 Exporting a namespace Member](#)

[16.5.6 Importing a Module to Use Its Exported Declarations](#)

[16.5.7 Example: Attempting to Access Non-Visible Module Contents](#)

[16.6 Global Module Fragment](#)

[16.7 Separating Interface from Implementation](#)

16.7.1 Example: The `:private` Module Fragment

16.7.2 Example: Module Implementation Units

16.7.3 Example: Modularizing a Class

16.8 Partitions

16.8.1 Example: Module Interface Partition Units

16.8.2 Module Implementation Partition Units

16.8.3 Example: “Submodules” vs. Partitions

16.9 Additional Modules Examples

16.9.1 Example: Importing the C++ Standard Library as Modules

16.9.2 Example: Cyclic Dependencies Are Not Allowed

16.9.3 Example: `imports` Are Not Transitive

16.9.4 Example: Visibility vs. Reachability

16.10 Modules Can Reduce Translation Unit Sizes and Compilation Times

16.11 Migrating Code to Modules

16.12 Future of Modules and Modules Tooling

16.13 Wrap-Up

Appendix: Modules Videos Bibliography

Appendix: Modules Articles Bibliography

Appendix: Modules Glossary

16.1 Introduction

20 Mod Modules—one of C++20’s “big four” features—provide a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details.¹ Each module is a uniquely named, reusable group of related declarations and definitions with a well-defined interface that client code can use.

¹. Daveed Vandevoorde, “Modules in C++ (Revision 6),” January 11, 2012. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf>.

This chapter presents many complete, working code examples that introduce modules. Modules help developers be more productive, especially as they build, maintain and evolve **large software systems**.² Modules also help make

such systems more scalable.³ C++ creator Bjarne Stroustrup says, “Modules offer a historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century).”⁴

2. Gabriel Dos Reis, “Programming in the Large With C++ 20 - Meeting C++ 2020 Keynote,” December 11, 2020. Accessed August 7, 2021. <https://www.youtube.com/watch?v=j4du4LNslI>.
3. Vassil Vassilev1, David Lange1, Malik Shahzad Muzaffar, Mircho Rodozov, Oksana Shadura and Alexander Penev, “C++ Modules in ROOT and Beyond,” August 25, 2020. Accessed August 15, 2021. <https://arxiv.org/pdf/2004.06507.pdf>.
4. Bjarne Stroustrup, “Modules and macros.” February 11, 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf>.

Even in small systems, modules can offer immediate benefits in every program. For example, you can replace C++ standard library `#include` preprocessor directives with `import` statements. This eliminates repeated processing of `#included` content because modules-style imported headers ([Section 16.4](#)), and modules in general, are **compiled once** then reused where you import them in the program.

Brief History of Modules

C++ modules were challenging to implement, and the effort took many years. The first designs were proposed in the early 2000s, but the C++ committee was focused on completing C++0x, which eventually became C++11. In 2014, modularization efforts continued with the proposal, “A Module System for C++,” based on Gabriel Dos Reis’s and Bjarne Stroustrup’s work at Texas A&M University.^{5,6} The C++ committee formed a modules study group that eventually led to the Modules TS (technical specification).⁷ Most subsequent committee discussions focused on technical compromises and the specification details for C++20’s modules capabilities.⁸

5. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—[Section 9.3.1](#) Modules,” June 12, 2020. Accessed August 14 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.
6. Gabriel Dos Reis, Mark Hall and Gor Nishanov, “A Module System for C++,” May 27, 2014. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf>.
7. Gabriel Dos Reis (Ed.), “Working Draft, Extensions to C++ for Modules,” January 29, 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4690.pdf>.

std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf.

8. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—[Section 9.3.1 Modules](#),” June 12, 2020. Accessed August 14 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.

Compiler Support for Modules

At the time of this writing, our preferred compilers did not fully support modules. We tried each live-code example on each compiler. For the most part, we were able to compile each example on each compiler. We present the required compilation commands for each example and indicate which compiler(s), if any, did not support that modules feature.⁹

9. The compilation steps for all three compilers might change as their modules implementations evolve. If you encounter problems with the steps, email us at deitel@deitel.com. We'll respond promptly and post updates on the book's website <https://deitel.com/c-plus-plus-20-for-programmers>.

Docker Containers for g++ 11.2 and clang++ 13

To test our modules code examples in the most current g++ and clang++ versions, we used the following free **Docker containers** from <https://hub.docker.com>:

- For g++, we used the official GNU GCC container's latest version (11.2):

```
docker pull gcc:latest
```

- The LLVM/Clang team does not have an official Docker container, but many working containers are available on hub.docker.com. We used the most recent and widely downloaded one containing clang++ version 13:¹⁰

10. The version of clang++ in Xcode does not have as many of the C++20 features implemented as the version directly from the LLVM/Clang team. Also, at the time of this writing, using "latest" rather than "13" in the docker pull command gives you a docker container with clang++ 12 not 13.

[Click here to view code image](#)

```
docker pull teeks99/clang-ubuntu:13
```

If you're not familiar with running Docker containers, see the book's Before You Begin section, and the Docker overview and getting started instructions

at:

[Click here to view code image](#)

<https://docs.docker.com/get-started/overview/>

C++ Compilation and Linking Prior to C++20

In [Section 16.2](#), we'll discuss the traditional C++ compilation and linking process and various problems with that model.

Advantages and Goals of Modules

In [Section 16.3](#), we'll point out various modules benefits, some of which correct problems with and improve upon the pre-C++20 compilation process.

Future of Modules and Modules Tooling

[Section 16.12](#), Future of Modules and Modules Tooling, discusses and provides references for some C++23 modules features that are under development.

Videos and Articles Bibliographies; Modules Glossary

For your further study, we provide appendices at the end of this chapter containing lists of videos, articles, papers and documentation we referenced as we wrote this chapter. We also provide a modules glossary with key modules terms and definitions.

16.2 Compilation and Linking Prior to C++20

C++ has always had a **modular architecture for managing code** via a combination of **header files** and **source-code files**:

- Under the guidance of **preprocessor directives**, the **preprocessor** performs **text substitutions** and other text manipulations on each **source-code file**. A preprocessed source-code file is called a **translation unit**.¹¹

¹¹. "Translation unit (programming)." Accessed August 18, 2021.
[https://en.wikipedia.org/wiki/Translation_unit_\(programming\)](https://en.wikipedia.org/wiki/Translation_unit_(programming)).

- The compiler converts each **translation unit** into an **object-code file**.
- The **linker** combines a program's **object-code files** with library object files, such as those of the **C++ standard library**, to create a program's

executable.

This approach has been around since the 1970s. C++ inherited it from C.¹² We discuss the preprocessor in online Appendix D, Preprocessor.¹³

12. Gabriel Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer,” October 5, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.

13. If you’re not familiar with the preprocessor, you might want to read online Appendix D, Preprocessor, before reading this chapter.

Problems with the Current Header-File/Source-Code-File Model

The preprocessor is simply a **text-substitution mechanism**—it does not understand C++. As motivation for C++20 modules, C++ creator Bjarne Stroustrup calls out three key preprocessor problems:¹⁴

14. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed August 14 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.

Err  • One header’s contents can affect any subsequent `#included` header, so **the order of `#includes` is important** and can cause subtle errors.

Err  • A given C++ entity can have **different declarations in multiple translation units**. The compiler and linker do not always report such problems.

Perf  **Perf**  • **Reprocessing the same `#included` content is slow**, particularly in large programs where a header can be included dozens or even hundreds of times. For each `#include`, the preprocessor will insert the header’s contents in a **translation unit**, possibly causing the same code to be compiled many times. Eliminating this reprocessing by instead `importing` headers as **header units** can significantly improve compilation times in large codebases.

Other preprocessor problems include:¹⁵

15. Bryce Adelstein Lelbach, “Modules are coming,” May 1, 2020. Accessed August 12, 2021. <https://www.youtube.com/watch?v=yee9i2rUF3s>.

- Definitions in headers can violate C++’s **One Definition Rule (ODR)**.

The ODR says, “No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template, default argument for a parameter (for a function in a given scope), or default template argument.”¹⁶

16. “C++20 Standard: 6 Basics—6.3 One-definition rule.” Accessed August 12, 2021. <https://eel.is/c++draft/basic.def.odr>.

- **Headers do not offer encapsulation**—everything in a header is available to the translation unit that `#includes` the header.
- Headers can have **accidental cyclic dependencies** on one another that cause compilation errors and other problems.¹⁷

17. “Circular dependency.” Accessed August 18, 2021. https://en.wikipedia.org/wiki/Circular_dependency.

 • Headers often define **macros—#define preprocessing directives** that create constants represented as symbols (such as, `#define SIZE 10`), and function-like operations (such as, `#define SQUARE(x) ((x) * (x))`).¹⁸ Macros are not part of C++, so the compiler cannot check their syntax and cannot report **multiple-definition errors** if two or more headers define the same macro name.

18. The all capital letters naming for preprocessor constants (like `SIZE`) and macros (like `SQUARE`) is a convention that some programmers prefer.

16.3 Advantages and Goals of Modules

Some modules benefits include:^{19,20,21,22,23}

19. Corentin Jabot, “What do we want from a modularized Standard Library?,” May 16, 2020. Accessed August 12, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2172r0.pdf>.

20. Gabriel Dos Reis and Pavel Curtis, “Modules, Componentization, and Transition,” October 5, 2015. Accessed August 12, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0141r0.pdf>.

21. Daniela Engert, “Modules the beginner’s guide,” May 2, 2020. Accessed August 6, 2021. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

22. Rainer Grimm, “C++20: The Advantages of Modules,” May 10, 2020. Accessed August 12, 2021. <https://www.modernescpp.com/index.php/cpp20-modules>.

23. Dmitry Guzeев, “A Few Words on C++ Modules,” January 8, 2018. Accessed August 10, 2021.

<https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64>.

 SE • Better organization and componentization of large codebases.

- Reducing translation unit sizes.

 Perf • Reduced compilation times.²⁴

24. Rene Rivera, “Are modules fast? (revision 1),” March 6 2019, Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1441r1.pdf>.

- Eliminating repetitive `#include` processing—a **compiled module does not have to be re-processed for every source code file that uses it**.
- Eliminating `#include` ordering issues.
- Eliminating many preprocessor directives that can introduce subtle errors.
- Eliminating **One Definition Rule (ODR)** violations.

Cons of Modules

Some modules disadvantages include:^{25,26}

25. Steve Downey, “Writing a C++20 Module,” July 5, 2021. Accessed August 18, 2021. <https://www.youtube.com/watch?v=A04piAqV9mg>.

26. Eric Niebler, “C++ Modules Might Be Dead-on-Arrival,” January 27, 2019. Accessed August 18, 2019. <https://vector-of-bool.github.io/2019/01/27/modules-doa.html>.

- Modules support is incomplete in most C++ compilers at the time of this writing.
- Existing codebases will need to be modified to fully benefit from modules.
- Modules impact systems at a level where it can be challenging to make modules portable across systems.
- Modules do not solve C++ problems with respect to packaging and distributing software conveniently, like the package managers used with several other popular languages.
- Compiled modules have compiler-specific aspects that are not portable across compilers and platforms, so you might need to distribute modules

as source-code until appropriate modules tooling becomes available.

- Modules uptake initially will be slow as organizations cautiously review the capabilities, decide how to structure new codebases, potentially modify existing ones and wait for information about the experiences of others.
- There currently are few modules recommendations and guidelines—for example, the C++ Core Guidelines have not yet been updated for modules.

16.4 Example: Transitioning to Modules—Header Units

SE  One goal of modules is to **eliminate the need for the preprocessor**. There are enormous numbers of preexisting libraries. Most libraries today are provided as

- **header-only libraries**,
- a combination of **headers and source-code files**, or
- a combination of **headers and object-code files**.

Mod  It will take time for library developers to modularize their libraries. Some libraries might never be modularized. As a transitional step from the preprocessor to modules, you can

import existing headers²⁷ from the **C++ standard library**, as shown in line 3 of Fig. 16.1. Doing so treats that header as a **header unit**.

27. “C++20 Standard: 10 Modules—10.3 Import declaration.” Accessed August 12, 2021.
<https://eel.is/c++draft/module.import>.

[Click here to view code image](#)

```
1 // fig16_01.cpp
2 // Importing a standard library header as a
3 import <iostream>; // instead of #include <
4
5 int main() {
```

```
6     std::cout << "Welcome to C++20 Modules!\";  
7 }
```



```
Welcome to C++20 Modules!
```

Fig. 16.1 | Importing a standard library header as a header unit.

How Header Units Differ from Header Files

Perf Rather than the preprocessor including the header's contents into a source-code file, the compiler processes the header as a **translation unit**, compiling it and producing information to treat the header as a module. In large-scale systems, this **improves compilation performance** because the **header** is compiled once, rather than having its contents inserted into every **translation unit** that includes the **header**.²⁸ Header units are similar to using **pre-compiled headers** in some C++ environments prior to modules.²⁹

28. Gabriel Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer,” October 5, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.

29. Cameron DaCamara, “Practical C++20 Modules and the future of tooling around C++ Modules,” May 4, 2020. Accessed August 13, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.

Unlike `#include` directives, the order of imports is irrelevant, so for example:³⁰

30. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed August 14, 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.

```
import SomeModule;
import SomeOtherModule;
```

produces the same results as:

```
import SomeOtherModule;
import SomeModule;
```

A **header unit**'s declarations are available to the **importing translation unit**

because **header units** implicitly “**export**” their contents.³¹ You’ll see in Section 16.5 that you can specify which declarations a module **exports** for use in other **translation units**—giving you precise control over your module’s interface. Unlike a `#include`, an `import` statement does not add code to the importing translation unit, so “an unused import is essentially costfree.”³² Also, preprocessor directives in a **translation unit** that appear before you `import` a **header unit** do not affect the header unit’s contents.³³

31. “C++20 Standard: 10 Modules—10.3 Import declaration.” Accessed August 12, 2021. <https://eel.is/c++draft/module.import>.

32. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed August 14 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.

33. Eric Niebler, “Understanding C++ Modules: Part 3: Linkage and Fragments,” October 7, 2019. Accessed August 15, 2021. <https://vector-of-bool.github.io/2019/10/07/modules-3.html>.

Import All Headers as Header Units (if Possible)

SE  You should generally `import` all headers as **header units** and use `#includes` only if necessary.³⁴ Unfortunately, not all headers can be processed as **header units**. You’ll typically use `#include` if:

34. Daniela Engert, “Modules the beginner’s guide,” May 2, 2020. Accessed August 6, 2021. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

Err  • importing a **header** as a **header unit** produces **compilation errors**, or
• you need to include a **header** that depends on **#defined preprocessor macros**—referred to as **preprocessor state**.

Compiling with Header Units in Microsoft Visual Studio

At the time of this writing, many modules features require a preview release of Visual Studio. You can install **Visual Studio preview** releases from:

[Click here to view code image](#)

<https://visualstudio.microsoft.com/vs/preview/>

On the **Available** tab, choose the preview release you wish to install, download the installer and run it. For this chapter, we used the free **Microsoft Visual Studio Community 2019 Preview version 16.11.0 Preview 3.0**, but you also may use the **Microsoft Visual Studio Community 2022 Preview 2** —or the latest preview release available.³⁵

³⁵. We tried the Visual Studio 2022 preview release and found compilation with it to be slow.

To compile any of this chapter's examples that use **header units**, configure your Visual Studio project as follows:³⁶

³⁶. This might change once C++20 modules are fully implemented in Visual C++.

1. Right-click the project name in **Solution Explorer** and select **Properties**.
2. In the **Property Pages** dialog, select **All Configurations** from the **Configurations** drop-down.
3. In the left column, under **Configuration Properties > C/C++ > Language**, set the **C++ Language Standard** option to **Preview - Features from the Latest C++ Working Draft (/std:c++latest)**.
4. In the left column, under **Configuration Properties > C/C++ > All Options** set the **Scan Sources for Module Dependencies** option to **Yes**.
5. Click **Apply**, then click **OK**.

Add `fig16_01.cpp` to your project's **Source Files** folder, then compile and run the project.

Compiling with Header Units in g++

In `g++`, you must first **compile each header** you'll **import** as a **header unit**.^{37,38,39} The following command compiles the `<iostream>` standard library header as a **header unit**:

³⁷. This might change once C++20 modules are fully implemented in `g++`.

³⁸. “3.23 C++ Modules.” Accessed August 10, 2021.
https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

³⁹. Nathan Sidwell, “C++ Modules: A Brief Tour,” October 19, 2020. Accessed August 10, 2021.
<https://accu.org/journals/overload/28/159/sidwell/>.

[Click here to view code image](#)

```
g++ -fmodules-ts -x c++-system-header iostream
```

- The **-fmodules-ts compiler flag** is currently required rather than `-std=c++20` to compile any code that uses C++20 modules.
- The **-x c++-system-header compiler flag** indicates that we are compiling a **C++ standard library header** as a **header unit**. You specify the header's name without the angle brackets.⁴⁰

⁴⁰. There are also `c++-header` and `c++-user-header` flags for precompiling other headers as header units. See https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

Next, compile the source-code file `fig16_01.cpp` using the following command:

[Click here to view code image](#)

```
g++ -fmodules-ts fig16_01.cpp -o fig16_01
```

This produces an **executable** named `fig16_01`, which you can execute with the command:

```
./fig16_01
```

Compiling with Header Units in clang++

Use the following command to compile this example in clang++ version 12:⁴¹

⁴¹. This might change once C++20 modules are fully implemented in clang++.

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-maps fig16_01.cpp -o fig16_01
```



This produces an **executable** named `fig16_01`, which you can execute with the command:

```
./fig16_01
```

In the compilation command, the flags have the following meanings:

- The flag `-std=c++20` indicates that we’re using C++20 language features.
- On some systems, `g++`’s C++ standard library is the default C++ library. The flag `-stdlib=libc++` ensures that `clang++`’s C++ standard library is used.
- The flags `-fimplicit-modules` and `-fimplicit-module-maps` enable `clang++` to generate and find the information it needs to treat **headers** as **header units**.

16.5 Example: Creating and Using a Module

 Next, let’s define our first **module**. A **module’s interface** specifies the module members that the module makes available for use in other **translation units**. You do this by **exporting the member’s declaration** using the **export** keyword in one of four ways:⁴²

⁴². “C++ Standard: 10 Modules—10.2 Export declaration.” Accessed August 12, 2021. <https://eel.is/c++draft/module.interface>.

- **export its declaration directly**,
- **export a block in braces ({ and })**—which may contain many declarations,
- **export a namespace**—which may contain many declarations, and
 - **export a namespace member**—which also exports the namespace’s name, but not the namespace’s other members.

Recall that if the first occurrence of an identifier is its definition, it also serves as the identifier’s declaration. So each item listed above may export declarations or definitions.

16.5.1 `module` Declaration for a Module Interface Unit

 Figure 16.2 defines our first **module unit**⁴³—a **translation unit** that is part of a module. When a module is composed of **one translation unit**, the **module unit** is commonly referred to simply as a **module**. The module in Fig. 16.2 contains four functions (lines 8–10, 14–16, 21–23 and 28–30) to demonstrate **exporting declarations** for use in other **translation**

units. In the following subsections, we'll discuss the **export** and **namespace** “wrappers” around the functions in lines 14–16, 21–23 and 28–30.

43. “C++ Standard: 10 Modules—10.1 Module units and purviews.” Accessed August 12, 2021.
<https://eel.is/c++draft/module.unit>.

[Click here to view code image](#)

```
1 // Fig. 16.2: welcome..hxx
2 // Primary module interface for a module named welcome
3 export module welcome; // introduces the module
4
5 import <string>; // class string is used in
6
7 // exporting a function
8 export std::string welcomeStandalone() {
9     return "welcomeStandalone function called"
10 }
11
12 // export block exports all items in the block
13 export {
14     std::string welcomeFromExportBlock() {
15         return "welcomeFromExportBlock function called"
16     }
17 }
18
19 // exporting a namespace exports all items in it
20 export namespace TestNamespace1 {
21     std::string welcomeFromTestNamespace1()
22     return "welcomeFromTestNamespace1 function called"
23 }
24
25
26 // exporting an item in a namespace exports it
27 export namespace TestNamespace2 {
28     export std::string welcomeFromTestNamespace2()
29 }
```

```
29         return "welcomeFromTestNamespace2 fun"
30     }
31 }
```

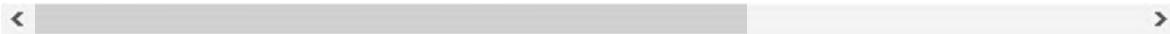


Fig. 16.2 | Primary module interface for a module named `welcome`.

Module Declaration and Module Naming

Mod Line 3's **module declaration** names the module `welcome`. **Module names** are identifiers separated by dots (.)⁴⁴—such as `deitel.time` or `deitel.math`, which we'll use in subsequent examples. The dots do not have special meaning—in fact, there's a proposal to remove dot-separated naming from modules.⁴⁵ Both `deitel.time` and `deitel.math` begin with “`deitel.`” but these modules are not “submodules” of a larger module named `deitel`. All declarations from the **module declaration** to the end of the **translation unit** are part of the **module purview**, as are declarations from all other units that make up the module.⁴⁶ We show **multi-file modules** in subsequent sections.

44. Corentin Jabot, “Naming guidelines for modules,” June 16, 2019. Accessed August 18, 2021. <https://isocpp.org/files/papers/P1634R0.html>.

45. Michael Spencer, “P1873R1: remove.dots.in.module.names,” September 17, 2019. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1873r1.html>.

46. “C++ Standard: 10 Modules—10.1 Module units and purviews.” Accessed August 12, 2021. <https://eel.is/c++draft/module.unit>.

Mod **Mod** When you precede a **module declaration** with **export**, it introduces a **module interface unit**, which specifies the **module members** that the **client code** can access. Every module has one **primary module interface unit** containing an **export module declaration** that introduces the module's name. You'll see in [Section 16.8](#) that the **primary module interface unit** may be composed of **module interface partition units**.

Module Interface File Extension

Microsoft Visual C++ uses the **.ixx filename extension** for **module**

interface units. To add a **module interface unit** to your Visual C++ project:

1. Right-click your project's **Source Files** folder and select **Add Module....**
2. In the **Add New Item** dialog, specify a file name (we used the name `welcome.ixxx`), specify where you want to save the file and click **Add**.
3. Replace the default code with the code in [Fig. 16.2](#).

You are not required to use the **.ixx filename extension**. If you name a **module interface unit**'s file with a different extension, right-click the file in your project, select **Properties**, and ensure that the file's **Item Type** is set to **C/C++ compiler**.

Module Interface File Extensions for g++ and clang++

The **g++** and **clang++** compilers do not require special filename extensions for **module interface units**. We'll show how to enable **g++** and **clang++** to compile `.ixx` files so you can use the same filename extensions for all three compilers.

Common filename extensions you'll encounter when using C++20 modules include:

- **.ixx**—Microsoft Visual C++ filename extension for the **primary module interface unit**.
- **.ifc**—Microsoft Visual C++ filename extension for the compiled version of the **primary module interface unit**.⁴⁷

⁴⁷ Cameron DaCamara, “Practical C++20 Modules and the future of tooling around C++ Modules,” May 4, 2020. Accessed August 13, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.

- **.cpp**—Filename extension for the C++ source code in a **translation unit**, including **module units**.
- **.cppm**—A recommended **clang++** filename extension for **module units**.
- **.pcm**—The **primary module interface unit** filename extension in **clang++**. When you compile a **primary module interface unit** in **clang++**, the resulting file uses this extension.

16.5.2 Exporting a Declaration

 You must **export a declaration** to make it available for use outside the module. Line 8 of Fig. 16.2 applies **export** to a function definition, which exports the **function's declaration** (that is, its **prototype**) as part of the **module's interface**. All exported declarations must appear after a **module declaration** in a **translation unit** and must appear at either **file scope** (known as **global namespace scope**) or in a **named namespace's scope** (Section 16.5.5). The declarations in **export** statements must not have **internal linkage**,⁴⁸ which includes:

48. “C++ Standard: 10 Modules—10.2 Export declaration.” Accessed August 12, 2021.
<https://eel.is/c++draft/module.interface>.

- static variables and functions at **global namespace scope** in a **translation unit**,
- const or constexpr **global variables** in a **translation unit**, and
- identifiers declared in “**unnamed namespaces**.”⁴⁹

49. “Namespaces—Unnamed namespaces.” Accessed August 13, 2021.
https://en.cppreference.com/w/cpp/language/namespace#Unnamed_names

Also, if a module defines any preprocessor macros, they're for use only in that module and cannot be **exported**.⁵⁰

50. Gabriel Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer,” October 5, 2019. Accessed August 6, 2021.
<https://www.youtube.com/watch?v=tjSuKOz5HK4>.

Defining Templates in Modules

Similar to headers, when you **define a template in a module**, you must **export** its complete definition so the compiler has access to it wherever the module is imported.

16.5.3 Exporting a Block of Declarations

   Rather than applying **export** to individual declarations, you can place a group of declarations in an **export block** (lines 13–17). **Every declaration in an export block is implicitly exported.** An **export block**'s braces do not define a new scope, so

identifiers declared in such a block continue to exist beyond the block's closing brace. An **export block** must be placed at **global namespace scope**; otherwise, a compilation error occurs.

16.5.4 Exporting a namespace

  A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will collide with a variable of the same name in a different scope, possibly creating a naming conflict and resulting in errors. C++ solves this problem with **name-spaces**. Each namespace defines a scope in which identifiers and variables are placed. As you know, the C++ standard library's features are defined in the **std namespace**, which helps ensure that these identifiers do not conflict with identifiers in your own programs. Placing `main` in a namespace is a compilation error.

Defining and Exporting namespaces

Lines 20–24 define the namespace `TestNamespace1`. A namespace's body is delimited by braces (`{ }`). A namespace may contain constants, data, classes and functions. Definitions of namespaces must be placed at **global namespace scope** or must be **nested** in other name-spaces. Unlike classes, namespace members may be defined in separate but identically named namespace blocks. For example, each C++ standard library header has a namespace block like:

[Click here to view code image](#)

```
namespace std {  
    // standard library header's declarations  
}
```

 indicating that the **header's declarations** are in namespace `std`. When you **export** a name-space, all its members are **exported**.

Accessing namespace Members

To use a **namespace** member, you must qualify the member's name with the **namespace name** and the **scope resolution operator** (`::`), as in the expression

[Click here to view code image](#)

```
TestNamespace1::welcomeInTestNamespace1()
```

or you must provide a **using declaration** or **using directive** before the member is used in the **translation unit**. A **using declaration** like

[Click here to view code image](#)

```
using TestNamespace1::welcomeInTestNamespace1;
```

brings one name (`welcomeInTestNamespace1`) into the scope where the directive appears. A **using directive** like

[Click here to view code image](#)

```
using namespace TestNamespace1;
```

SE  brings all the names from the specified namespace into the scope where the directive appears. Members of the same namespace can access one another directly without using a **namespace qualifier**.

16.5.5 Exporting a **namespace** Member

Mod  It's also possible to **export** specific **namespace** members rather than an entire **namespace**, as shown in lines 27–31. In this case, the **namespace**'s name also is exported. This does not implicitly export the **namespace**'s other members.

16.5.6 Importing a Module to Use Its Exported Declarations

Mod  **SE**  To use a **module's exported declarations** in a given **translation unit**, you must provide an **import declaration** (Fig. 16.3, line 4) at **global namespace scope** containing the module's name. The **module's exported declarations** are available from the **import** declaration to the end of the **translation unit**. **Importing a module does not insert the module's code into a translation unit**. So, unlike headers, modules do not need **include guards** (Section 9.7.4). Lines 7–10 call the four functions we exported from the `welcome` module (Fig. 16.2). For the functions defined in namespaces, lines 9 and 10 precede each function name with its

namespace's name and the **scope resolution operator** (`::`). The program's output shows that we could call each of module `welcome`'s exported functions.

[Click here to view code image](#)

```
1 // fig16_03.cpp
2 // Importing a module and using its exported items
3 import <iostream>;
4 import welcome; // import the welcome module
5
6 int main() {
7     std::cout << welcomeStandalone() << '\n'
8         << welcomeFromExportBlock() << '\n'
9         << TestNamespace1::welcomeFromTestNamespace1()
10        << TestNamespace2::welcomeFromTestNamespace2()
11 }
```



```
welcomeStandalone function called
welcomeFromExportBlock function called
welcomeFromTestNamespace1 function called
welcomeFromTestNamespace2 function called
```

Fig. 16.3 | Importing a module and using its exported items.

Importing a Module into Another Module

Err Sometimes one module will import another to use (and possibly re-export) its features. In such cases, the `import` statement must appear after the importing module's **module declaration** and before any of the **importing module's declarations and definitions**; otherwise, a compilation error occurs.

Compiling This Example in Visual C++

In Visual C++, ensure that `fig16_03.cpp` is in your project's **Source**

Files folder, then run your project to compile the module and the main application.

Compiling This Example in g++

In g++, execute the following commands, which might change once C++20 modules are finalized in this compiler:

1. Compile the `<string>` and `<iostream>` headers as **header units** because they're used in our module and our main application, respectively:

[Click here to view code image](#)

```
g++ -fmodules-ts -x c++-system-header string  
g++ -fmodules-ts -x c++-system-header iostream
```

2. Compile the **module interface unit**—this produces the file `welcome.o`:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ welcome.ixx
```

3. Compile the main application and link it with `welcome.o`—this command produces the **executable file** `fig16_03`:

[Click here to view code image](#)

```
g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03
```

In Step 2:

- the `-c` option says to compile `welcome.ixx`, but not link it, and
- the `-x c++` option indicates that `welcome.ixx` is a C++ file.

The `-x c++` is required because `.ixx` is not a standard g++ filename extension. If we name `welcome.ixx` as `welcome.cpp`, then the `-x c++` option is not required.

Compiling This Example in clang++

In clang++, execute the following commands, which might change once C++20 modules are finalized in this compiler:

1. Compile the **module interface unit** into a **precompiled module (.pcm) file**, which is specific to the clang++ compiler:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-maps -x c++ welcome.ixx  
-Xclang -emit-module-interface -o welcome.pcm
```



2. Compile the main application and link it with welcome.pcm—this command produces the **executable file** fig16_03:

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-maps -fprebuilt-module-path=fig16_03.cpp welcome.pcm -o fig16_03
```



In Step 1:

- the `-c` option says to compile `welcome.ixx`, but not link it, and
- the `-x c++` option indicates that `welcome.ixx` is a C++ file.

The `-x c++` is required because `.ixx` is not a standard clang++ filename extension. If we name the file `welcome.cpp`, then the `-x c++` option is not required. In Step 2, the option

```
-fprebuilt-module-path=.
```

indicates where clang++ can locate **precompiled module (.pcm) files**—the dot (.) is the current folder but could be a relative or full path to another location on your system.

16.5.7 Example: Attempting to Access Non-Visible Module Contents

   Modules do not implicitly **export declarations**—this is known as **strong encapsulation**. You have precise control over the declarations you export for use in other **translation units**. Figure 16.4 defines a **primary module interface unit** (line 3) named

`deitel.math` containing namespace `deitel::math` that is not exported. You can add this file to your Visual C++ project, as you saw in [Section 16.5.1](#). In the namespace, we define two functions—`square` (lines 7–9) is exported, and `cube` (lines 12–14) is not. **Exporting the function `square` implicitly exports the enclosing namespace's name but does not export the namespace's other members.** Since `cube` is not exported, other **translation units** cannot call it (as you'll see in [Fig. 16.5](#)). This is a key difference from **headers**—everything declared in a **header** can be used wherever you `#include` it.⁵¹

⁵¹. Gabriel Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer,” October 5, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.

[Click here to view code image](#)

```
1 // Fig. 16.4: deitel.math..hxx
2 // Primary module interface for a module named deitel.math
3 export module deitel.math; // introduces the module
4
5 namespace deitel::math {
6     // exported function square; namespace deitel::math
7     export int square(int x) {
8         return x * x;
9     }
10
11    // non-exported function cube is not imported by other modules
12    int cube(int x) {
13        return x * x * x;
14    }
15 }
```



Fig. 16.4 | Primary module interface for a module named `deitel.math`. (Part 1 of 2.)

SE A It's good practice in a module to **put exported identifiers in**

namespaces to help **avoid name collisions** when multiple **modules export the same identifier**. We found in our research that **namespace names** typically mimic their **module names**⁵²—so for the `deitel.math` module, we specified the namespace `deitel::math` (line 5).

52. Daniela Engert, “Modules the beginner’s guide,” May 2, 2020. Accessed August 6, 2021. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

Err In Fig. 16.5, line 4 imports the `deitel.math` module. Line 9 calls the module’s exported `deitel::math::square` function, **qualifying the function name with its enclosing namespace’s name**. This compiles because `add` was exported by the `deitel.math` module. Line 12, however, results in compilation errors—the `deitel.math` module did not export the `cube` function. The compilation errors in Fig. 16.5 are from Visual C++. We highlighted key error messages in bold and added vertical spacing for readability.

[Click here to view code image](#)

```
1 // fig16_05.cpp
2 // Showing that a module's non-exported ide:
3 import <iostream>;
4 import deitel.math; // import the welcome m
5
6 int main() {
7     // can call square because it's exported
8     // which implicitly exports the namespaces
9     std::cout << "square(3) = " << deitel::ma
10
11    // cannot call cube because it's not exp
12    std::cout << "cube(3) = " << deitel::ma
13 }
```

< >

```
Build started...
1>----- Build started: Project: modules_demo, Configuration: Debug Win32 -----
1>Scanning sources for module dependencies...
```

```
1>deitel.math.ixx
1>fig16_05.cpp
1>Compiling...
1>deitel.math.ixx
1>fig16_05.cpp

1>C:\Users\pauldeitel\Documents\examples\ch16\fi
05\fig16_05.cpp(12,47): error C2039: 'cube': is
1>C:\Users\pauldeitel\Documents\examples\example
itel.math.ixx(5): message : see declaration of 'cube'

1>C:\Users\pauldeitel\Documents\examples\example
05\fig16_05.cpp(12,51): error C3861: 'cube': id
1>Done building project "modules_demo.vcxproj" -
=====
Build: 0 succeeded, 1 failed, 0 up-to-date
```

Fig. 16.5 | Showing that a module's non-exported identifiers are inaccessible. (Part 1 of 2.)

g++ Error Messages

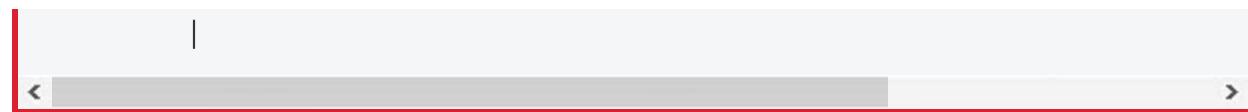
Err To see the g++ error messages, execute the following commands, each of which we explained in [Section 16.5.6](#):

1. g++ -fmodules-ts -x c++-system-header iostream
2. g++ -fmodules-ts -c -x c++ deitel.math.ixx
3. g++ -fmodules-ts fig16_05.cpp deitel.math.o

We highlighted the key error message in bold:

[Click here to view code image](#)

```
fig16_05.cpp: In function 'int main()':
fig16_05.cpp:12:49: error: 'cube' is not a member of
12 | std::cout << "cube(e) = " << deitel::math::cube(e);
```



clang++ Error Messages

 To see the clang++ error messages, execute the following commands, each of which we explained in [Section 16.5.6](#):

1. clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules

[Click here to view code image](#)

```
-fimplicit-module-maps -xc++ deitel.math..hxx  
-Xclang -emit-module-interface -o deitel.math.pcm
```

2. clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules

[Click here to view code image](#)

```
-fimplicit-module-maps -fprebuilt-module-path=.  
fig16_05.cpp deitel.math.pcm -o fig16_05
```

We highlighted the key error message in bold:

[Click here to view code image](#)

```
fig16_05.cpp:12:49: error: no member named 'cube' :  
    std::cout << "cube(e) = " << deitel::math::cube  
                           ~~~~~^  
1 error generated.
```

16.6 Global Module Fragment

As we mentioned, some **headers** cannot be compiled as **header units** because they require **preprocessor state**, such as macros defined in your **translation unit** or other **headers**. Such **headers** can be #included for use in a **module unit** by placing them in the **global module fragment**:⁵³

53. “C++ Standard: 10 Modules—10.4 Global module fragment,” Accessed August 10, 2021.

```
https://eel.is/c++draft/module.global.frag.
```

```
module;
```

which you place at the beginning of a **module unit**, before the **module declaration**. The **global module fragment** may contain only **#include headers**. A **module interface unit** can **export a declaration** that was **#included** into the **global module fragment**, so other **implementation units** that **import the module** can use that declaration.⁵⁴

54. Gabriel Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer," October 5, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.

Mod Any declaration **#included** in the **global module fragment** but not used by the module is **discarded** by the compiler.⁵⁵ **Global module fragments** from all **module units** are placed into the **global module**, as is non-modularized code in other **non-module translation units**, such as the one containing function `main`.

55. Eric Niebler, "Understanding C++ Modules: Part 3: Linkage and Fragments—'Discarded' Declarations," October 7, 2019. Accessed August 13, 2021. <https://vector-of-bool.github.io/2019/10/07/modules-3.html>.

16.7 Separating Interface from Implementation

In Chapters 9 and 10, we defined classes using **.h headers** and **.cpp source-code files** to **separate a class's interface from its implementation**. The same approach is commonly used in **function libraries**—a **.h header** specifies a **library's function prototypes**, and a corresponding **.cpp file** provides their **implementations**.

Modules support **separating interface from implementation** but with more flexibility than **headers** and **source-code files**. You can:

SE  • **define both interface and implementation in the same source file** (as shown in this section) **while still hiding the implementation details from client code**, or

SE  • **split your interface and implementation into separate files** (shown in subsequent sections).

16.7.1 Example: The :private Module Fragment

Let's demonstrate a module with its **interface** and **implementation** defined separately in one **translation unit**. Figure 16.6 defines a **primary module interface unit** named `deitel.math` (line 4) that exports its `deitel::math` namespace. The namespace contains the **prototype** for an average function (line 11) that receives a reference to a `const vector<int>` and returns the average of the `vector<int>`'s values as a double.

[Click here to view code image](#)

```
1 // Fig. 16.6: deitel.math..hxx
2 // Primary module interface for a module named
3 // with a :private module fragment.
4 export module deitel.math; // introduces the module
5
6 import <numeric>;
7 import <vector>;
8
9 export namespace deitel::math {
10     // calculate the average of a vector<int>
11     double average(const std::vector<int>& v);
12 }
13
14 module :private; // private implementation
15
16 namespace deitel::math {
17     // average function's implementation
18     double average(const std::vector<int>& v) {
19         double total{std::accumulate(values.begin(), values.end(),
20             return total / values.size();
21     }
22 }
```

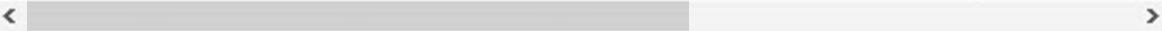


Fig. 16.6 | Primary module interface for a module named

deitel.math with a `:private` module fragment.

Mod Line 14 introduces the **:private module fragment** for **separating a module's implementation details from its interface** in the **primary module interface unit**. With this approach, the **primary module interface unit** must be the module's only **module unit**. The contents of the **:private module fragment** do not affect the **module's interface**, so this fragment may not export declarations.⁵⁶

56. “C++20 Standard: 10 Modules—Private module fragment.” Accessed August 11, 2021. <https://eel.is/c++draft/module.private.frag>.

SE  Perf  The key benefit of a **primary module interface unit** containing a **:private module fragment** is that you can conveniently define both a **module's interface and its implementation details** in **one translation unit** without exposing the implementation details to other **translation units**. Changes to the implementation details in a **:private module fragment** do not affect other **translation units** that import the module.⁵⁷ So, they do not need to be recompiled, potentially improving compilation times.⁵⁸ Line 7 in Fig. 16.7 imports the deitel.math module, and line 17 uses the module's `deitel::math::average` function to calculate the average of vector integers' values.

57. “C++20 Standard: 10 Modules—Private module fragment.” Accessed August 11, 2021. <https://eel.is/c++draft/module.private.frag>.

58. This will probably require build tools that are smart enough to recognize that the module interface did not change.

[Click here to view code image](#)

```
1 // fig16_07.cpp
2 // Using the deitel.math module's average f·
3 import <algorithm>;
4 import <iostream>;
5 import <iterator>;
6 import <vector>;
7 import deitel.math; // import the deitel.ma·
```

```
8
9  int main() {
10    std::ostream_iterator<int> output(std::cout);
11    std::vector integers{1, 2, 3, 4};
12
13    std::cout << "vector integers: ";
14    std::copy(integers.begin(), integers.end(),
15              output);
16    std::cout << "\naverage of integer's elements: ";
17    deitel::math::average(integers) <<
18 }
```

< >

```
vector integers: 1 2 3 4
average of integer's elements: 2.5
```

Fig. 16.7 | Using the `deitel.math` module's `average` function.

Compiling This Example in Visual C++

Add the `deitel.math.ixx` file to your Visual C++ project as you did in [Section 16.5.1](#), then run your project to compile the module and the **main application**.

Compiling This Example in g++

First, use the commands introduced in [Section 16.5.6](#) to compile the **standard library headers** `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as **header units**. Then compile⁵⁹ the **primary module interface unit** as follows:

59. At the time of this writing, `g++` does not support the `:private` module fragment. Once it does, you should be able to use the specified commands to compile this example.

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Finally, **compile the main application and link it** with `deitel.math.o` as follows:

[Click here to view code image](#)

```
g++ -fmodules-ts fig16_07.cpp deitel.math.o -o fig16_07
```

Compiling This Example in clang++

In clang++, compile the **primary module interface unit** into a **precompiled module (.pcm) file** as follows:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-interface -fimplicit-module-maps -xc++ deitel.math..hxx -Xclang -emit-module-interface -o deitel.math.]
```

Then **compile the main application and link it** with deitel.math.pcm as follows:

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-interface -fimplicit-module-maps -fprebuilt-module-path=fig16_07.cpp deitel.math.pcm -o fig16_07
```

16.7.2 Example: Module Implementation Units

 **Modularization** Sometimes it's useful to break a larger module into smaller, more manageable pieces—for example, when a team of developers is working on different aspects of the same module. You can **split a module definition into multiple source files**. Let's reimplement the program of [Section 16.7.1](#) using a **primary module interface unit** for the **module's interface** and a separate source code file for the **module's implementation details**.

Primary Module Interface Unit

The deitel.math module's **primary module interface unit** ([Fig. 16.8](#)) exports the deitel::math namespace (lines 7–10), containing the average function's prototype. Note that the function's definition has been

removed from this file.

[Click here to view code image](#)

```
1 // Fig. 16.8: deitel.math.ixx
2 // Primary module interface for a module named
3 export module deitel.math; // introduces the module
4
5 import <vector>;
6
7 export namespace deitel::math {
8     // calculate the average of a vector<int>
9     double average(const std::vector<int>& v);
10}
```



Fig. 16.8 | Primary module interface for a module named deitel.math.

Module Implementation Unit

   All files containing **module declarations** without the **export keyword** (line 3 of Fig. 16.9) are **module implementation units**.⁶⁰ These can be used to **split larger modules into multiple source files to make the code more manageable**. **Module implementation units** are typically placed in .cpp files. Line 3 indicates that this file is a **module implementation unit** for the deitel.math module. A **module implementation unit** implicitly imports the interface for the specified module name. The compiler combines the **primary module interface unit** and its corresponding **module implementation unit(s)** into a single **named module**⁶¹ that other **translation units** can import. Lines 10–13 implement the average function in a namespace that must have the same name as the one enclosing average's **function prototype** in the **primary module interface unit** (Fig. 16.8). The main program in this example is identical to Fig. 16.7 and produces the same output, so we did not repeat it in this section.

60. “C++20 Standard: 10 Modules—10.1 Module units and purviews.” Accessed August 12, 2021.
<https://eel.is/c++draft/module#unit>.
61. “C++20 Standard: 10 Modules.” Accessed August 12, 2021.
<https://eel.is/c++draft/module>.

[Click here to view code image](#)

```
1 // Fig. 16.9: deitel.math-impl.cpp
2 // Module implementation unit for the module
3 module deitel.math; // this file's contents
4
5 import <numeric>;
6 import <vector>;
7
8 namespace deitel::math {
9     // average function's implementation
10    double average(const std::vector<int>& values) {
11        double total{std::accumulate(values.begin(), values.end(),
12            return total / values.size();
13    }
14 }
```



Fig. 16.9 | Module implementation unit for the module `deitel.math`.

Compiling This Example in Visual C++

Add the `deitel.math.iwx` file to your **Visual C++ project**, as you did in [Section 16.5.1](#). Ensure that your project includes in its **Source Files** folder

- `deitel.math.iwx`—the **primary module interface unit**,
 - `deitel.math-impl.cpp`⁶²—the **module implementation unit**, and
62. We named the module implementation unit with “-impl” in the filename primarily to support the compilation steps of the g++ compiler. This is not a requirement.
- `fig16_07.cpp`—a copy of the **main application file** from [Section 16.7.1](#).

Then, simply run your project to compile the module and the main application.

[Compiling This Example in g++](#)

In g++, use the commands introduced in [Section 16.5.6](#) to compile the **standard library headers** `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as **header units**. Next, compile the **primary module interface unit** as follows:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Then, compile the **module implementation unit** as follows:

[Click here to view code image](#)

```
g++ -fmodules-ts -c deitel.math-impl.cpp
```

We now have files named `deitel.math.o` and `deitel.math-impl.o` representing the **module's interface and implementation**.

Finally, **compile the main application and link it** with `deitel.math.o` and `deitel.math-impl.o` as follows:

[Click here to view code image](#)

```
g++ -fmodules-ts fig16_07.cpp deitel.math.o deitel.math-impl.o  
-o fig16_07
```

[Compiling This Example in clang++](#)

In clang++, compile the **primary module interface unit** into a **precompiled module (.pcm) file** as follows:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-interface  
-fimplicit-module-maps -xc++ deitel.math.ixx  
-Xclang -emit-module-interface -o deitel.math.j
```

Next, compile the **module implementation unit** into an **object file** as

follows:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-maps -fmodule-file=deitel.math-impl.cpp
```



The option `-fmodule-file=deitel.math.pcm` specifies the name of the module's **primary module interface unit**. Finally, **compile the main application and link it** with `deitel.math-impl.o` and `deitel.math.pcm` as follows:

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-maps -fprebuilt-module-path=deitel.math-impl.o deitel.math.pcm -o fig16_07
```



16.7.3 Example: Modularizing a Class

So far, we've demonstrated only **function definitions in modules**. Let's define a simplified version of Chapter 9's Time class with its **interface** in a **primary module interface unit** and its **implementation** in a **module implementation unit**. We'll name our module `deitel.time` and place the class in the `deitel::time` namespace.

[deitel.time Primary Module Interface Unit](#)

The `deitel.time` module's **primary module interface unit** (Fig. 16.10) defines and exports namespace `deitel::time` (lines 7–19) containing the Time class definition (lines 8–18).

[Click here to view code image](#)

```
1 // Fig. 16.10: deitel.time.ixx
2 // Primary module interface for a simple Time class
3 export module deitel.time; // declare primary module
4
```

```
5 import <string>; // rather than #include <s
6
7 export namespace deitel::time {
8     class Time {
9         public:
10            // default constructor because it can
11            explicit Time(int hour = 0, int minute
12
13            std::string toString() const;
14        private:
15            int m_hour{0}; // 0 - 23 (24-hour clo
16            int m_minute{0}; // 0 - 59
17            int m_second{0}; // 0 - 59
18        };
19    }
```



Fig. 16.10 | Primary module interface for a simple Time class.

deitel.time Module Implementation Unit

The **module declaration** in line 4 of Fig. 16.11 indicates that deitel.time-impl.cpp is the **deitel.time module implementation unit**. The rest of the file defines class Time's member functions. Line 8:

[Click here to view code image](#)

```
using namespace deitel::time;
```

gives this **module implementation unit** access to namespace deitel::time's contents. However, any **translation unit** that imports the deitel.time module does not see this **using directive**. So other **translation units** still must access our **module's exported names** with deitel::time or via their own using statements.

[Click here to view code image](#)

```
1 // Fig. 16.11: deitel.time-impl.cpp
2 // deitel.time module implementation unit c
3 // Time class member function definitions.
4 module deitel.time; // module implementation
5
6 import <stdexcept>;
7 import <string>;
8 using namespace deitel::time;
9
10 // Time constructor initializes each data m
11 Time::Time(int hour, int minute, int second)
12     // validate hour, minute and second
13     if ((hour < 0 || hour >= 24) || (minute
14         (second < 0 || second >= 60)) {
15         throw std::invalid_argument{
16             "hour, minute or second was out of
17         }
18
19     m_hour = hour;
20     m_minute = minute;
21     m_second = second;
22 }
23
24 // return a string representation of the Ti
25 std::string Time::toString() const {
26     using namespace std::string_literals;
27
28     return "Hour: "s + std::to_string(m_hour
29         "\nMinute: "s + std::to_string(m_minu
30         "\nSecond: "s + std::to_string(m_seco
31 }
```



Fig. 16.11 | deitel.time module implementation unit containing the Time class member function definitions.

Using Class Time from the `deitel.time` Module

The program in Fig. 16.12 imports the `deitel.time` module (line 7) and uses class `Time`. For convenience, line 8 indicates that this program uses the module's `deitel::time` namespace, but you also can fully qualify the mention of class `Time` (lines 11 and 17), as in:

```
deitel::time::Time
```

[Click here to view code image](#)

```
1 // fig16_12.cpp
2 // Importing the deitel.time module and using its Time class
3 import <iostream>;
4 import <stdexcept>;
5 import <string>;
6
7 import deitel.time;
8 using namespace deitel::time;
9
10 int main() {
11     const Time t{12, 25, 42}; // hour, minute, second
12
13     std::cout << "Time t:\n" << t.toString()
14
15     // attempt to initialize t2 with invalid values
16     try {
17         const Time t2{27, 74, 99}; // all bad
18     }
19     catch (const std::invalid_argument& e) {
20         std::cout << "t2 not created: " << e.what()
21     }
22 }
```



```
Time t:
Hour: 12
Minute: 25
```

```
Second: 42  
t2 not created: hour, minute or second was out of range
```

Fig. 16.12 | Importing the `deitel.time` module and using the modularized `Time` class.

Compiling This Example in Visual C++⁶³

Add `deitel.time.i.hxx` to your **Visual C++ project**, as in [Section 16.5.1](#). Next, ensure that your project includes in its **Source Files** folder

- `deitel.time.i.hxx`—the **primary module interface unit**,
- `deitel.time-impl.cpp`⁶³—the **module implementation unit**, and

63. We named the module implementation unit with "-impl" in the filename primarily to support the compilation steps of the g++ compiler. This is not a module implementation unit requirement.

- `fig16_12.cpp`—the **main application file**.

Then, simply run your project to compile the module and the **main application**.

Compiling This Example in g++⁶⁴

In g++, use the commands introduced in [Section 16.5.6](#) to compile the **standard library headers** `<iostream>`, `<string>` and `<stdexcept>` as **header units**.

Next, compile the **primary module interface unit**:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.time.i.hxx
```

Then, compile the **module implementation unit** as follows—this command assumes that the `{fmt}` library is located in the `libraries` folder within the book's examples folder:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.time-impl.i.hxx
```

We now have files named `deitel.time.o` and `deitel.time-impl.o` representing the `deitel.time` **module's interface and implementation**.

Finally, compile the **main application** source file and link it with `deitel.time.o` and `deitel.time-impl.o`:

[Click here to view code image](#)

```
g++ -fmodules-ts fig16_12.cpp deitel.time.o deitel.time-impl.o  
-o fig16_07
```

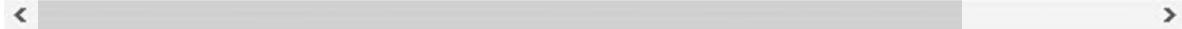


Compiling This Example in clang++

In `clang++`, compile the **primary module interface unit** into a **precompiled module (.pcm) file**:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-maps  
-fimplicit-module-maps -xc++ deitel.time.ii  
-Xclang -emit-module-interface -o deitel.time.ij
```



Next, compile the **module implementation unit** into an **object file**:

[Click here to view code image](#)

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-module-maps  
-fimplicit-module-maps -fmodule-file=deitel.time.ij  
deitel.time-impl.cpp
```



The option `-fmodule-file=deitel.math.pcm` specifies the name of the module's **primary module interface unit**. Finally, compile `fig16_12.cpp` and link it with `deitel.math-impl.o` and `deitel.math.pcm`:

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-maps  
-fimplicit-module-maps -fprebuilt-module-path=fig16_12.cpp  
deitel.time-impl.o deitel.time.pcm
```



16.8 Partitions

  You can divide a **module's interface** and/or its **implementation** into smaller pieces called **partitions**.⁶⁴ When working on large projects, this can help you organize a module's components into smaller, more manageable **translation units**. Breaking a larger module into smaller **translation units** also can **reduce compilation times** in large systems. Only **translation units** that have changed and **translation units** that depend on those changes would need to be recompiled.⁶⁵ Whether items are recompiled would be **determined by the compiler's modules tooling**.⁶⁶ The compiler aggregates a **module's partitions** into a single **named module** for import into other **translation units**.

⁶⁴. At the time of this writing, clang++ does not yet support partitions.

⁶⁵. Cameron DaCamara, "Practical C++20 Modules and the future of tooling around C++ Modules," May 4, 2020. Accessed August 12, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.

⁶⁶. Modules tooling is under development at the time of this writing and will continue to evolve over the next several years.

16.8.1 Example: Module Interface Partition Units

In this example, we'll create a `deitel.math` module that exports four functions in its **primary module interface unit**—`square`, `cube`, `squareRoot` and `cubeRoot`. We'll divide these functions into two **module interface partition units** (`powers` and `roots`) to show partition syntax. Then we'll aggregate their exported declarations into a single **primary module interface partition**.

`deitel.math:powers` Module Interface Partition Unit

 Line 3 of Fig. 16.13 indicates that the **translation unit** `deitel.math-powers.iixx` is a **module interface partition unit**. The notation `deitel.math:powers` specifies that the **partition name** is "powers", and the partition is part of the module `deitel.math`. This **module interface partition** exports the `deitel::math` namespace, containing the functions `square` and `cube`. **Module partitions are not visible outside their module, so they cannot be imported into translation**

units that are not part of the same module.⁶⁷

67. "Modules—Module Partitions." Accessed August 13, 2021.
<https://en.cppreference.com/w/cpp/language/modules>.

Click here to view code image

```
1 // Fig. 16.13: deitel.math-powers..hxx
2 // Module interface partition unit deitel.math::powers
3 export module deitel.math:powers;
4
5 export namespace deitel::math {
6     double square(double x) { return x * x; }
7     double cube(double x) { return x * x * x; }
8 }
```



Fig. 16.13 | Module interface partition unit
deitel.math:additive.

deitel.math:roots Module Interface Partition Unit

Line 3 of [Fig. 16.14](#) indicates that the **translation unit** deitel.math-roots.hxx is a **module interface partition unit** with the **partition name** is "roots". The partition belongs to module deitel.math. This partition exports the deitel::math namespace, containing the functions squareRoot and cubeRoot. There are several **rules to keep in mind for partitions**:



- All module interface partitions with the same module name are part of the same module (in this case, deitel.math). They are not implicitly known to one another, and they do not implicitly import the module's interface.⁶⁸

68. Gabriel Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer," October 5, 2019. Accessed August 6, 2021.
<https://www.youtube.com/watch?v=tjSuKOz5HK4>.



- Partitions may be imported only into other module units

that belong to the same module.

 • One **module interface partition unit** can **import** another from the same module to use the other partition's features.

[Click here to view code image](#)

```
1 // Fig. 16.14: deitel.math-roots.ixx
2 // Module interface partition unit deitel.math:roots;
3 export module deitel.math:roots;
4
5 import <cmath>;
6
7 export namespace deitel::math {
8     double squareRoot(double x) { return std::sqrt(x); }
9     double cubeRoot(double x) { return std::cbrt(x); }
10}
```



Fig. 16.14 | Module interface partition unit deitel.math:multiplicative.

deitel.math Primary Module Interface Unit

Figure 16.15 defines the deitel.math.ixx **primary module interface unit**. Every module must have one **primary module interface unit** with an **export module declaration** that does not include a **partition name** (line 4). Lines 8 and 9 import and export the **module interface partition units**:

- Each **import** is followed by a colon (:) and the name of a **module interface partition unit** (in this case, powers or roots).
- Placing the **export** keyword before **import** indicates that each **module interface partition unit**'s exported members also should be part of the deitel.math module's **primary module interface**.

 **SE**  The users of your module cannot see its partitions.⁶⁹

69. “C++ Standard: 10 Modules—10.1 Module units and purviews.” Accessed August 12, 2021.
<https://eel.is/c++draft/module.unit>.

[Click here to view code image](#)

```
1 // Fig. 16.15: deitel.math.ixx
2 // Primary module interface unit deitel.math
3 // the module interface partitions :powers and :roots
4 export module deitel.math; // declares the module
5
6 // import and re-export the declarations in
7 // interface partitions :powers and :roots
8 export import :powers;
9 export import :roots;
```

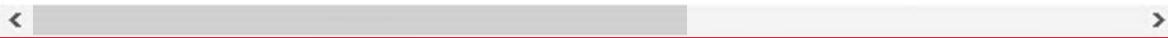


Fig. 16.15 | Primary module interface unit that exports declarations from the module interface partitions :powers and :roots.

You also can **export import primary module interface units**. Let’s assume we have modules named A and B. If module A’s **primary module interface unit** contains

```
export import B;
```

then a **translation unit** that imports A also imports B and can use its **exported declarations**.

If you **export import a header unit**, its preprocessor macros are available for use only in the **importing translation unit**—they are not re-exported. So, to use a macro from a **header unit** in a specific **translation unit**, you must explicitly **import** the header.

Using the `deitel.math` Module

Figure 16.16 imports the `deitel.math` module (line 4) and lines 9–12 use its **exported functions** to demonstrate that the **primary module interface** contains all the functions exported by the **module interface partitions**.

[Click here to view code image](#)

```
1 // fig16_16.cpp
2 // Using the deitel.math module's functions
3 import <iostream>;
4 import deitel.math; // import the deitel.ma
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5)
11    << "\nsquareRoot(9): " << squareRoot(
12        << "\ncubeRoot(1000): " << cubeRoot(1
13    }
```

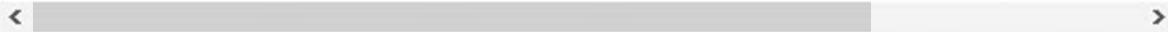


Fig. 16.16 | Using the `deitel.math` module's functions.

Compiling This Example in Visual C++

Add the files `deitel.math-powers.ixx` `deitel.math-roots.ixx` and `deitel.math.ixx` to your **Visual C++ project** using the steps from [Section 16.5.1](#), then add the file `fig16_16.cpp` to the project's **Source Files** folder. Run your project to compile the module and the **main application**.

Compiling This Example in g++

When **building a module with partitions**, you must build the partitions before the primary **module interface unit**. In `g++`, use the commands introduced in [Section 16.5.6](#) to compile the **standard library headers** `<cmath>` and `<iostream>` as **header units**. Next, compile each **module**

interface partition unit:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.math-powers..hxx  
g++ -fmodules-ts -c -x c++ deitel.math-roots..hxx
```

Then, compile the **primary module interface unit**:

[Click here to view code image](#)

```
g++ -fmodules-ts -c -x c++ deitel.math..hxx
```

Finally, compile the **main application** and the `{fmt}` library's `format.cc` source file and link it with `deitel.math-additive.o`, `deitel.math-multiplicative.o` and `deitel.math.o`:

[Click here to view code image](#)

```
g++ -fmodules-ts -I ../../libraries/fmt/include f:  
../../libraries/fmt/src/format.cc deitel.math-  
deitel.math-roots.o deitel.math.o -o fig16_16
```



16.8.2 Module Implementation Partition Units

  You also can divide **module implementations** into **module implementation partition units** to define a module's implementation details across **multiple source-code files**.⁷⁰ Again, this can help you organize a module's components into smaller, more manageable **translation units** and possibly **reduce compilation times** in large systems. In a **module implementation partition**, the **module declaration** must not contain `export` keyword:

⁷⁰. Richard Smith, “Merging Modules—Section 2.2 Module Partitions,” February 22, 2019. Accessed August 13, 2021. <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf>.

```
module ModuleName:PartitionName;
```

Module implementation partition units may not `export` declarations and do not implicitly import the **primary module interface**.⁷¹ At the time of this writing, none of our preferred compilers support **module**

implementation partitions, so we do not show them here.

[71. Daniela Engert, “Modules the beginner’s guide,” May 2, 2020. Accessed August 13, 2021.
`https://www.youtube.com/watch?v=Kqo-jIq4V3I`.](#)

16.8.3 Example: “Submodules” vs. Partitions

SE  Some libraries, like the **C++ standard library**, are quite large. Programmers using such a library might want the flexibility to import only portions of it. A library vendor can divide a library into **logical “submodules,”** each with its own **primary module interface unit**. These can be imported independently. In addition, library vendors can provide a **primary module interface unit** that aggregates the “submodules” by importing and re-exporting their interfaces. Let’s reimplement [Section 16.8](#)’s `deitel.math` module using only **primary module interface units** to demonstrate the flexibility this provides to users.

`deitel.math.powers` Primary Module Interface Unit

First, let’s rename `deitel.math-powers.ixx` as `deitel.math.powers.ixx`. We used the convention “*-name*” previously to indicate that the `powers` partition was part of module `deitel.math`. In this program, `deitel.math.powers` is a **primary module interface unit** ([Fig. 16.17](#)). Rather than declaring a **module interface partition**, as in [Fig. 16.13](#):

[Click here to view code image](#)

```
export module deitel.math:powers;
```

line 3 of [Fig. 16.17](#) declares a **primary module interface unit** with a dot-separated name:

[Click here to view code image](#)

```
export module deitel.math.powers;
```

We can now independently import `deitel.math.powers` and use its functions ([Fig. 16.18](#)).

[Click here to view code image](#)

```
1 // Fig. 16.17: deitel.math.powers.ixx
2 // Primary module interface unit deitel.math
3 export module deitel.math.powers;
4
5 export namespace deitel::math {
6     double square(double x) { return x * x;
7     double cube(double x) { return x * x * x
8 }
```



Fig. 16.17 | Primary module interface unit deitel.math.powers.

[Click here to view code image](#)

```
1 // fig16_18.cpp
2 // Using the deitel.math.powers module's fu
3 import <iostream>;
4 import deitel.math.powers; // import the de
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5) << '\n';
11 }
```



```
square(6): 36
cube(5): 125
```

Fig. 16.18 | Using the deitel.math.powers module's functions.

deitel.math.roots Primary Module Interface Unit

Next, let's rename the file deitel.math-roots.ixx as deitel.math.roots.ixx and make deitel.math.powers a

primary module interface unit (Fig. 16.19). Rather than declaring a **module interface partition**, as in Fig. 16.14:

[Click here to view code image](#)

```
export module deitel.math:roots;
```

line 3 of Fig. 16.19 declares a **primary module interface unit** with a dot-separated name:

[Click here to view code image](#)

```
export module deitel.math.roots;
```

We can now independently import deitel.math.roots and use its functions (Fig. 16.20).

[Click here to view code image](#)

```
1 // Fig. 16.19: deitel.math.roots.ixx
2 // Primary module interface unit deitel.math.roots
3 export module deitel.math.roots;
4
5 import <cmath>;
6
7 export namespace deitel::math {
8     double squareRoot(double x) { return std::sqrt(x); }
9     double cubeRoot(double x) { return std::cbrt(x); }
10 }
```



Fig. 16.19 | Primary module interface unit deitel.math.roots.

[Click here to view code image](#)

```
1 // fig16_20.cpp
2 // Using the deitel.math.roots module's functions
3 import <iostream>;
4 import deitel.math.roots; // import the deitel.math.roots module
```

```
5
6     using namespace deitel::math;
7
8     int main() {
9         std::cout << "squareRoot(9): " << squareRoot(9)
10            << "\ncubeRoot(1000): " << cubeRoot(1000)
11    }
```

< >

```
square(6): 36
cube(5): 125
```

Fig. 16.20 | Using the `deitel.math.roots` module's functions. (Part 1 of 2.)

`deitel.math` Primary Module Interface Unit

SE A Figures 16.17 and 16.19 are now **separate modules**—even though their dot-separated names `deitel.math.powers` and `deitel.math.roots` indicate a **logical relationship** between them, and both modules export the `deitel::math` namespace. For convenience, we can now **aggregate these separate modules in a primary module interface unit** that **exports imports** both “**submodules**” as shown in lines 7 and 8 of Fig. 16.21. We can then use all the functions from both “**submodules**” by importing `deitel.math` (Fig. 16.22).

SE A With these “**submodules**” developers now have the flexibility to:

- import `deitel.math.powers` to use only `square` and `cube`,
- import `deitel.math.roots` to use only `squareRoot` and `cubeRoot`, or
- import the aggregated module `deitel.math` to use all four functions.

[Click here to view code image](#)

```
1 // Fig. 16.21: deitel.math.hxx
```

```
2 // Primary module interface unit deitel.math
3 // from "submodules" deitel.math.powers and
4 export module deitel.math; // primary module
5
6 // import and re-export deitel.math.powers
7 export import deitel.math.powers;
8 export import deitel.math.roots;
```



Fig. 16.21 | Primary module interface unit deitel.math aggregates declarations from "submodules" deitel.math.powers and deitel.math.roots.

[Click here to view code image](#)

```
1 // fig16_22.cpp
2 // Using the deitel.math module's functions
3 import <iostream>;
4 import deitel.math; // import the deitel.math module
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5)
11    << "\nsquareRoot(9): " << squareRoot(9)
12    << "\ncubeRoot(1000): " << cubeRoot(1000)
13 }
```



```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

Fig. 16.22 | Using the deitel.math module's functions. (Part 2 of 2.)

16.9 Additional Modules Examples

The next several subsections demonstrate additional modules concepts including:

- importing the modularized Microsoft and clang++ standard libraries,
- some module restrictions and the compilation errors you'll receive if you violate those restrictions, and
- the difference between module members that other translation units can use by name vs. module members that other translation units can use indirectly.

16.9.1 Example: Importing the C++ Standard Library as Modules

Perf  The C++ standard does not currently require compilers to provide a **modularized standard library**. Microsoft provides one for Visual C++, which they split into several modules, and clang++ has a single-module version of the standard library. You can import the following modules into your Visual C++ projects and “potentially speed up compilation times depending on the size of your project”:⁷²

⁷². Colin Robertson and Nick Schonning, “Overview of modules in C++,” December 13, 2019. Accessed August 13, 2021. <https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160>.

- std.core—This module contains most of the standard library, except for the items below.
- std.filesystem—Module containing the `<filesystem>` header’s capabilities.
- std.memory—Module containing the `<memory>` header’s capabilities.
- std.regex—Module containing the `<regex>` header’s capabilities.
- std.threading—Module containing the capabilities of all the concurrency-related headers: `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` and `<thread>`.

For clang++, you can import the entire C++ standard library with:

```
import std;
```

SE A In Visual C++, you cannot combine code that **#includes standard library headers** with code that **imports Microsoft's standard library modules**. The compiler will not know which version of the standard libraries to choose when the final executable is linked. At the time of this writing, g++ did not have a **modularized standard library**.

Figure 16.23 imports the **std.core module** (line 3) then uses cout from the standard library's <iostream> capabilities to output a string.

[Click here to view code image](#)

```
1 // Fig. 16.23.cpp
2 // Importing Microsoft's modularized standard library
3 import std.core; // provides access to most
4
5 int main() {
6     std::cout << "Welcome to C++20 Modules!\\";
7 }
```



```
Welcome to C++20 Modules!
```

Fig. 16.23 | Importing Microsoft's modularized standard library.

Compiling the Program in Visual C++

Compiling a program that uses **Microsoft's modularized standard library** requires additional project settings, which may change once Microsoft's modules implementation is finalized. First, you must ensure that C++ modules support is installed:

1. In Visual Studio, select **Tools > Get Tools and Features...**
2. In the **Individual Components** tab, ensure that **C++ Modules for v142 build tools** is checked. If not, check it, then click **Modify** to install it. You will need to close Visual Studio to complete

the installation.

Next, you must configure several project settings:

1. In the **Solution Explorer**, right-click your project and select **Properties** to open the **Property Pages** dialog.
2. In the left column, select **Configuration Properties > C/C++ > Code Generation**.
3. In the right column, ensure that **Enable C++ Exceptions** is set to **Yes (/EHsc)**.
4. In the right column, ensure that **Runtime Library** is set to **Multi-threaded DLL (/MD)** if you are compiling in **Release** mode or **Multi-threaded Debug DLL (/MDd)** if you are compiling in **Debug** mode. You can choose settings for **Release** or **Debug** mode by changing the value in the **Configurations** drop-down at the top of the **Property Pages** dialog.
5. In the left column, select **Configuration Properties > C/C++ > Language**.
6. In the right column, ensure that **Enable Experimental C++ Standard Library Modules** is set to **Yes (/experimental:module)** and that **C++ Language Standard** is set to **Preview - Features from the Latest C++ Working Draft (/std:c++latest)**.

You can now compile and run Fig. 16.23.

Modifying and Compiling the Program in clang++

To compile this program in clang++, change line 3 of Fig. 16.23 to:

```
import std;
```

Then compile the program using the following command:

[Click here to view code image](#)

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-module-maps fig16_23.cpp -o fig16_23
```



16.9.2 Example: Cyclic Dependencies Are Not Allowed

A **module** is not allowed to have a dependency on itself—that is, a module cannot import itself directly or indirectly.^{73,74} Figures 16.24 and 16.25 define **primary module interface units** moduleA and moduleB—moduleA imports moduleB and vice versa. Each module indirectly has a dependency on itself due to the `import` statements in line 5 of Fig. 16.24 and line 5 of Fig. 16.25—moduleA imports moduleB which, in turn, imports moduleA.

73. “C++ Standard: 10 Modules—10.3 Import declaration.” Accessed August 14, 2021. <https://eel.is/c++draft/module.import>.

74. Eric Niebler, “Understanding C++ Modules: Part 2: export, import, visible, and reachable,” March 31, 2019. Accessed August 14, 2021. <https://vector-of-bool.github.io/2019/03/31/modules-2.html>.

[Click here to view code image](#)

```
1 // Fig. 16.24: moduleA.ixx
2 // Primary module interface unit that imports moduleB
3 export module moduleA; // declares the primary module
4
5 export import moduleB; // import and re-export moduleB
```

Fig. 16.24 | Primary module interface unit that imports moduleB.

[Click here to view code image](#)

```
1 // Fig. 16.25: moduleB.ixx
2 // Primary module interface unit that imports moduleA
3 export module moduleB; // declares the primary module
4
5 export import moduleA; // import and re-export moduleA
```

Fig. 16.25 | Primary module interface unit that imports moduleA.

 Compiling Figs. 16.24 and 16.25 in Visual C++ results in the following error message:

[Click here to view code image](#)

```
error : Cannot build the following source files because of
cyclic dependency between them: ch16\fig16_24-26\i
on ch16\fig16_24-26\moduleB.ixx depends on ch16\f
moduleA.ixx.
```

< >

You cannot compile this example in `g++` or `clang++` because each compiler requires a **primary module interface unit** to be compiled before you can import it. Since each module depends on the other, this is not possible.

16.9.3 Example: imports Are Not Transitive

 In Section 16.5.7, we mentioned that modules have **strong encapsulation** and **do not export declarations implicitly**. Thus, **import statements are not transitive**—if a **translation unit** imports A and A imports B, the **translation unit** that imported A does not automatically have access to B’s exported members.

Consider `moduleA` (Fig. 16.26) and `moduleB` (Fig. 16.27)—**moduleB imports but does not re-export moduleA** (line 6 of Fig. 16.27). As a result, **moduleA’s exported cube function is not part of moduleB’s interface**.

[Click here to view code image](#)

```
1 // Fig. 16.26: moduleA.ixx
2 // Primary module interface unit that exports
3 export module moduleA; // declares the primary
4
5 export int cube(int x) { return x * x * x;
```

< >

Fig. 16.26 | Primary module interface unit that exports function cube.

[Click here to view code image](#)

```
1 // Fig. 16.27: moduleB.ixx
2 // Primary module interface unit that imports
3 // moduleB and exports function square.
4 export module moduleB; // declares the primary
5
6 import moduleA; // import but do not export
7
8 export int square(int x) { return x * x; }
```



Fig. 16.27 | Primary module interface unit that imports, but does not export, moduleB and exports function square.

Err **Figure 16.28** imports moduleB and attempts to use moduleA's cube function (line 8), which generates errors because cube's declaration is not imported. The key error messages produced by Visual C++, g++ and clang++, respectively, are:

- error C3861: 'cube': identifier not found
- error: 'cube' was not declared in this scope
- error: declaration of 'cube' must be imported from module 'moduleA' before it is required

[Click here to view code image](#)

```
1 // fig16_28.cpp
2 // Showing that moduleB does not implicitly
3 import <iostream>;
4 import moduleB;
5
6 int main() {
```

```
7     std::cout << "square(6): " << square(6)
8         << "\ncube(5): " << cube(5) << '\n';
9 }
```



Fig. 16.28 | Showing that moduleB does not implicitly export moduleA's function.

16.9.4 Example: Visibility vs. Reachability

Mod^塊 Every example so far in which we used a **name exported from a module** demonstrated the concept of **visibility**. A **declaration is visible in a translation unit if you can use its name**. As you've seen, **any name exported from a module can be used in translation units that import the module**.

Mod^塊 Some declarations are **reachable** but not **visible**,^{75,76,77} meaning you **cannot explicitly mention the declaration's name in another translation unit, but the declaration is indirectly accessible**. Anything **visible** is **reachable**, but not vice versa. The easiest way to understand this concept is with code. To demonstrate **reachability**, we modified Fig. 16.10's **primary module interface unit** `deitel.time.ixx`.⁷⁸ There are two key changes (Fig. 16.29):

75. “C++ Standard: 10 Modules—10.7 Reachability.” Accessed August 14, 2021. <https://eel.is/c++draft/module.reach>.

76. Eric Niebler, “Understanding C++ Modules: Part 2: export, import, visible, and reachable,” March 31, 2019. Accessed August 14, 2021. <https://vector-of-bool.github.io/2019/03/31/modules-2.html>.

77. Richard Smith, “Merging Modules—Section 2.2 Module Partitions,” February 22, 2019. Accessed August 13, 2021. <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf>.

78. The module's implementation unit (Fig. 16.11) is identical for this example, so we do not show it here.

- We no longer export namespace `deitel::time` (line 7), so **class Time is not exported** and thus **not visible to translation units that import deitel.time**.

- We added an **exported** function `getTime` (line 21) that **returns a Time object to its caller**—we'll use this function's return value to demonstrate **reachability**.

[Click here to view code image](#)

```

1 // Fig. 16.29: deitel.time.ixx
2 // Primary module interface unit for the deitel.time module
3 export module deitel.time; // declare the primary module
4
5 import <string>; // rather than #include <string>
6
7 namespace deitel::time {
8     class Time { // not exported
9         public:
10            // default constructor because it can
11            explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13            std::string toString() const;
14        private:
15            int m_hour{0}; // 0 - 23 (24-hour clock)
16            int m_minute{0}; // 0 - 59
17            int m_second{0}; // 0 - 59
18        };
19
20        // exported function returns a valid Time object
21        export Time getTime() { return Time(6, 45, 30); }
22    }

```



Fig. 16.29 | Primary module interface unit for the `deitel.time` module.

The program in [Fig. 16.30](#) imports the `deitel.time` module (line 5). Line 10 calls the module's exported `getTime` function to get a `Time` object. Note that **we infer variable t's type**. If you were to replace `auto` in line 10

with **Time**, you'd get an error like the following (in Visual C++):

[Click here to view code image](#)

```
error C2065: 'Time': undeclared identifier
```

Mod because **Time** is not visible in this translation unit. However, Time's declaration is **reachable** because **getTime** returns a **Time** object—the compiler knows this, so it can infer variable's t's type. **When a declaration is reachable, the declaration's members become visible.** So, even though `deitel.time` does not export class `Time`, this translation unit can still call `Time` member function `toString` (line 14) to get t's string representation. The compilation steps for this program are the same as those in [Section 16.7.3](#), except that the main program's filename is now `fig16_30.cpp`.

[Click here to view code image](#)

```
1 // fig16_30.cpp
2 // Showing that type deitel::time::Time is
3 // and its public members are visible.
4 import <iostream>;
5 import deitel.time;
6
7 int main() {
8     // initialize t with the object returned :
9     // as type Time because the type is not :
10    auto t{ deitel::time::getTime() };
11
12    // Time's toString function is reachable
13    // class Time was not exported by module
14    std::cout << "Time t:\n" << t.toString()
15 }
```



```
Time t:
```

```
Hour: 6
```

Minute: 45
Second: 0

Fig. 16.30 | Showing that type `deitel::time::Time` is reachable and its public members are visible.

16.10 Modules Can Reduce Translation Unit Sizes and Compilation Times

One way in which modules can reduce compilation times is by eliminating repeated preprocessing of the same header files across many translation units in the same program. Consider the following simple program

[Click here to view code image](#)

```
#include <iostream>

int main() {
    std::cout << "Welcome to C++20 Modules! \n";
}
```

which has fewer than 90 characters, including the vertical spacing and indentation. When you compile this program, the preprocessor inserts the contents of `<iostream>` into the translation unit. Each of our preferred compilers has a flag that enables you to see the result of preprocessing a source-code file:

- `-E` in `g++` and `clang++`
- `/P` in Visual C++.

We used these flags to preprocess the preceding program. The **preprocessed translation unit sizes** on our system were:

- 1,023,010 bytes in `g++`,
- 1,883,270 bytes in `clang++`, and
- 1,497,116 bytes in Visual C++.

The preprocessed translation units are approximately 11,000 to 21,000 times the size of the original source file—a massive increase in the number

of bytes per translation unit. Now imagine a large project with **thousands of translation units** that each `#include` **many headers**. Every `#include` must be preprocessed to form the translation units that the compiler then needs to process.

Perf  **When you use header units, each header is processed only once as a translation unit.** Also, importing a **header unit does not insert the header's contents into each translation unit.** Together, these greatly reduce compilation times.

References

For more information on translation unit sizes and compilation performance, see the following resources:

- **Bjarne Stroustrup**, “**Thriving in a Crowded and Changing World: C++ 2006– 2020—Section 9.3.1 Modules**,” June 12, 2020. Accessed August 14, 2021. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.
- **Cameron DaCamara**, “**Practical C++20 Modules and the future of tooling around C++ Modules**,” May 4, 2020. Accessed August 13, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
- **Rainer Grimm**, “**C++20: The Advantages of Modules**,” May 10, 2020. Accessed August 5, 2021. <https://www.modernescpp.com/index.php/cpp20-modules>.
- **Rene Rivera**, “**Are modules fast? (revision 1)**,” March 6 2019, Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1441r1.pdf>

Compiler Profiling Tools

For tools you can use to profile compilation performance, see the following resources:

- Kevin Cadieux, Helena Gregg and Colin Robertson, “Get started with C++ Build Insights,” November 3, 2019. Accessed August 16, 2021. <https://docs.microsoft.com/en-us/cpp/build/>

<https://msvcbuildinsights.dev/microsoft.com/insights/get-started-with-cpp-build-insights?view=msvc-160&viewFallbackFrom=vs-2019>.

- “Clang 13 Documentation: Target-independent compilation options – `-ftime-report` and `-ftime-trace`,” Accessed August 16, 2021. <https://clang.llvm.org/docs/ClangCommandLineReference/IndependentCompilationOptions.html>.
- “Profiling the C++ compilation process.” Accessed August 16, 2021. <https://stackoverflow.com/questions/13559818/profiling-the-c-compilation-process>

16.11 Migrating Code to Modules

We’ve frequently referred to the C++ Core Guidelines for advice and recommendations on the proper ways to use various language elements. At the time of this writing, modules technology is still new, the popular compilers’ modules implementations are not complete, and the C++ Core Guidelines have not yet been updated with modules recommendations. There also are not many articles and videos discussing developers’ experiences with migrating existing software systems to modules. Some of the best are listed below. The Cameron DaCamara (Microsoft) and Steve Downey (Bloomberg) videos provide the most recent tips, guidelines and insights. The Daniela Engert and Nathan Sidwell videos each demonstrate modularizing existing code, and the Yuka Takahashi, Oksana Shadura and Vassil Vassilev paper discusses their experiences with modularizing portions of the large CERN ROOT C++ codebase:

- **Cameron DaCamara, “Moving a project to C++ named Modules,”** August 10, 2021. Accessed August 16, 2021. <https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/>.
- **Steve Downey, “Writing a C++20 Module,”** July 5, 2021. Accessed August 18, 2021. <https://www.youtube.com/watch?v=A04piAqV9mg>.
- **Daniela Engert, “Modules: The Beginner’s Guide,”** May 2, 2020. Accessed August 6, 2021. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

- **Yuka Takahashi, Oksana Shadura and Vassil Vassilev**, “**Migrating large code-bases to C++ Modules**,” August 22, 2019. Accessed August 16, 2021. <https://arxiv.org/abs/1906.05092>.
- **Nathan Sidwell**, “**Converting to C++20 Modules**,” October 4, 2019. Accessed August 14, 2021. <https://www.youtube.com/watch?v=KVvSWIEw3TTw>.

We will post additional resources as they become available at:

[Click here to view code image](#)

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

16.12 Future of Modules and Modules Tooling

The C++ standard committee has begun its work on C++23 for which one of the key items will be a **modular standard library**.⁷⁹ C++20 modules are so new that the tooling to help you use modules tooling is under development and will continue to evolve over several years. You’ve already seen some tooling. For example, if you used Visual C++ in this chapter’s examples, you saw that Visual Studio enables you to add modules to your projects, and its build tools can compile and link your modularized applications.

⁷⁹. “To boldly suggest an overall plan for C++23,” November 25, 2019. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html>

Many popular programming languages have module systems or similar capabilities:

• Java has the Java Platform Module System (JPMS) for which we wrote a chapter in our Java books and published an article in Oracle’s Java Magazine.⁸⁰

⁸⁰. Paul Deitel, “Understanding Java 9 Modules,” *Oracle Java Magazine*, September/October 2017. <https://www.oracle.com/a/ocom/docs/corporate/java-magazine-sept-oct-2017.pdf>.

• Python has a well-developed module system, which we used extensively in our Python books.^{81,82}

⁸¹. Paul Deitel and Harvey Deitel, *Python for Programmers*, 2019. Pearson Education, Inc.

⁸². Paul Deitel and Harvey Deitel, *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*, 11/e, 2020. Pearson Education, Inc.

- Microsoft's .NET platform languages like C# and Visual basic can modularize code using assemblies.

Wikipedia lists several dozen languages with modules capabilities.⁸³

83. “Modular programming.” Accessed August 18, 2021. https://en.wikipedia.org/wiki/Modular_programming.

Many languages provide tooling to help you work with their modules systems and modularize your code. Some tooling that might eventually appear in the C++ ecosystem include:

- **module-aware build tools** that manage compiling software systems,
- **tools to produce cross-platform module interfaces** so developers can distribute a module interface description and object code, rather than source code,
- **dependency-checking tools** to ensure that required modules are installed,
- **module discovery tools** to determine which modules and versions are installed,
- **tools that visualize module dependencies**, showing you the relationships among modules in software systems,
- **module packaging and distribution tools** to help developers install modules and their dependencies conveniently across platforms,
- and more.

References

A July 2021 paper from Daniel Ruoso of Bloomberg⁸⁴ discusses various problems with code reuse and build systems today and is meant to encourage discussions regarding the future of C++ modules tooling. That paper and the other resources below list in reverse chronological order various C++ standard and third-party vendor opportunities for module-aware tools that will improve the C++ development process: nm,./

84. Daniel Ruoso, “Requirements for Usage of C++ Modules at Bloomberg,” July 12, 2021. Accessed August 18, 2021. <https://isocpp.org/files/papers/P2409R0.pdf>.

- **Daniel Ruoso, “Requirements for Usage of C++ Modules at Bloomberg,”** July 12, 2021. Accessed August 18, 2021.

<https://isocpp.org/files/papers/P2409R0.pdf>.

- **Nathan Sidwell**, “**P1184: A Module Mapper**,” July 10, 2020. Accessed August 16, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1184r2.pdf>
- **Rob Irving, Jason Turner** and **Gabriel Dos Reis**, “**Modules Present and Future**,” June 18, 2020. Accessed August 16, 2021. <https://cppcast.com/modules-gaby-dos-reis/>.
- **Cameron DaCamara**, “**Practical C++20 Modules and the future of tooling around C++ Modules**,” May 4, 2020. Accessed August 12, 2021. Accessed August 16, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
- **Nathan Sidwell**, “**C++ Modules and Tooling**,” October 4, 2018. Accessed August 16, 2021. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.
- **Gabriel Dos Reis**, “**Modules Are a Tooling Opportunity**,” October 16, 2017. Accessed August 16, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0822r0.pdf>

16.13 Wrap-Up

In this chapter, we introduced modules—one of C++20’s new “big-four” features. You saw that modules help you organize your code, precisely control which declarations you expose to client code, and encapsulate implementation details. We discussed the advantages of modules, including how they make developers more productive, make systems more scalable, and can reduce translation unit sizes and improve build times. We also pointed out some disadvantages.

The chapter presented many complete, working modules code examples. You saw that even small systems can benefit from modules technology by transitioning from using preprocessor `#include` directives to importing standard library headers as header units. We created custom modules. We implemented a primary module interface unit to specify a module’s client-code interface, then imported that module into an application to use its exported members. We employed namespaces to avoid naming conflicts with other modules’ contents. You saw that non-exported members are not

accessible by name in `import`-ing translation units.

We separated interface from implementation—first in a primary module interface unit by placing the implementation code in the `:private` module fragment, then by using a module implementation unit. We divided a module into partitions to organize its components into smaller, more manageable translation units. We showed that “submodules” are more flexible than partitions because partitions cannot be `imported` into translation units that are not part of the same module.

We demonstrated how easy it is to `import` with a single statement either Microsoft’s or `clang++`’s modularized standard library. We showed that cyclic module dependencies are not allowed and that `imports` are not transitive. We discussed the difference between visible declarations and reachable declarations, mentioning that anything visible is reachable, but not necessarily vice versa.

We provided resources with tips for migrating legacy code to modules—a subject of great interest to organizations considering deploying modules technology. Finally, we discussed the future of C++20 modules and some types of modules tooling that might appear in the next several years. In the following appendices, for your further study we provide lists of the videos, articles, technical papers and documentation that we referenced as we wrote this chapter. We also include a glossary with key modules terms and definitions.

Modules technology is important. Once supported by the right tools, modules will provide C++ developers with significant opportunities to improve the design, implementation and evolution of libraries and large-scale software systems.

But modules are new and few organizations have experience using them. Some organizations will modularize new software—but what about the four decades’ worth of non-modularized legacy C++ software. Some of it will eventually be modularized. Some will never be, possibly because the people who built and understand the systems have moved on.

At Deitel & Associates, we work with many widely used programming languages. Based on our experience studying, writing about and teaching the Java Platform Module System (JPMS), for example, we believe the uptake on C++20 modules will be gradual. Java introduced JPMS in 2017. Compiler

vendor JetBrains' 2021 developer survey showed that 72% of Java developers are still working in some capacity with Java 8—the version of Java before JPMS was introduced.⁸⁵ Organizations using C++ will likely proceed with caution, too. Many will wait to learn about other organizations' experiences modularizing large legacy codebases and launching new modularized software-development projects.

85. "The State of Developer Ecosystem 2021." July 15, 2021. Accessed August 18, 2021. <https://www.jetbrains.com/lp/devcosystem-2021/>.

Chapter 17 presents C++20's concurrency and parallelism capabilities, including the standard library's parallel algorithms and multithreading features. We also introduce coroutines—the last of C++20's “big four” features.

Appendix: Modules Videos Bibliography

Videos are listed in reverse chronological order.

- **Steve Downey, “Writing a C++20 Module,”** July 5, 2021. Accessed August 18, 2021. <https://www.youtube.com/watch?v=AO4piAqV9mg>.
- **Daniela Engert, “The Three Secret Spices of C++ Modules,”** July 1, 2021. Accessed August 15, 2021. https://www.youtube.com/watch?v=l_83lyxWGtE.
- **Sy Brand, “C++ Modules: Year 2021,”** May 6, 2021. Accessed August 6, 2021. <https://www.youtube.com/watch?v=YcZntyWpqVQ>
- **Gabriel Dos Reis, “Programming in the Large With C++ 20 - Meeting C++ 2020 Keynote,”** December 11, 2020. Accessed August 7, 2021. <https://www.youtube.com/watch?v=j4du4LNsLiI>.
- **Marc Gregoire, “C++20: An (Almost) Complete Overview,”** September 26, 2020. Accessed August 18, 2021. <https://www.youtube.com/watch?v=FRkJCvHWdwQ>.
- **Cameron DaCamara, “Practical C++20 Modules and the future of tooling around C++ Modules,”** May 4, 2020. Accessed August 12, 2021. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
- **Timur Doumler, “How C++20 Changes the Way We Write Code,”** October 10, 2020. Accessed July 1, 2021. <https://www.youtube.com/watch?v=KJLjwXHgkY>.

<https://www.youtube.com/watch?v=ImLFLjSveM>.

- **Daniela Engert**, “**Modules: The Beginner’s Guide**,” May 2, 2020. Accessed August 6, 2021. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.
- **Bryce Adelstein Lelbach**, “**Modules are coming**,” May 1, 2020. Accessed August 7, 2020. <https://www.youtube.com/watch?v=yee9i2rUF3s>.
- **Pure Virtual C++ 2020 Conference**, April 30, 2020. Accessed August 14, 2021. <https://www.youtube.com/watch?v=c1ThUFISDF4>
- “**Demo: C++20 Modules**,” March 30, 2020. Accessed July 28, 2021. <https://www.youtube.com/watch?v=6SKIUeRaIZE>.
- **Daniela Engert**, “**Dr Module and Sister #include**,” December 5, 2019. Accessed August 15, 2021. <https://www.youtube.com/watch?v=OCFOTle2G-A>.
- **Boris Kolpackov**, “**Practical C++ Modules**,” Oct. 18, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=szHV6RdQdg8>.
- **Michael Spencer**, “**Building Modules**,” October 6, 2019. Accessed August 15, 2021. <https://www.youtube.com/watch?v=L0SHHkBenss>.
- **Gabriel Dos Reis**, “**Programming with C++ Modules: Guide for the Working Software Developer**,” October 5, 2019. Accessed August 6, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.
- **Gabriel Dos Reis**, “**Programming with C++ modules**,” October 5, 2019. Accessed August 7, 2021. <https://www.youtube.com/watch?v=tjSuKOz5HK4>.
- **Nathan Sidwell**, “**Converting to C++20 Modules**,” October 4, 2019. Accessed August 14, 2021. <https://www.youtube.com/watch?v=KVswIEw3TTw>.
- **Gabriel Dos Reis**, “**C++ Modules: What You Should Know**,” September 13 2019. Accessed August 15, 2021. <https://www.youtube.com/watch?v=MP6SJEBt6Ss>

- **Richárd Szalay**, “**The Rough Road Towards Upgrading to C++ Modules**,” June 16, 2019. Accessed August 15, 2021. <https://www.youtube.com/watch?v=XJxQs8qgn-c>.
- **Nathan Sidwell**, “**C++ Modules and Tooling**,” October 4, 2018. Accessed August 15, 2021. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.

Appendix: Modules Articles Bibliography

Articles are listed in reverse chronological order.

- **Cameron DaCamara**, “**Moving a project to C++ named Modules**,” August 10, 2021. Accessed August 15, 2021. <https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/>.
- **Cameron DaCamara**, “**Using C++ Modules in MSVC from the Command Line Part 1: Primary Module Interfaces**,” July 21, 2021. Accessed August 4, 2021. <https://devblogs.microsoft.com/cppblog/using-cpp-modules-in-msvc-from-the-command-line-part-1/>.
- **Daniel Ruoso**, “**Requirements for Usage of C++ Modules at Bloomberg**,” July 12, 2021. Accessed August 18, 2021. <https://isocpp.org/files/papers/P2409R0.pdf>.
- **Andreas Fertig** (book), “**Programming with C++20 — Concepts, Coroutines, Ranges, and more**.” Copyright 2021. <https://andreasfertig.info/books/programming-with-cpp20/>.
- **Nathan Sidwell**, “**C++ Modules: A Brief Tour**,” October 19, 2020. Accessed August 10, 2021. <https://accu.org/journals/overload/28/159/sidwell>.
- **Cameron DaCamara**, “**Standard C++20 Modules support with MSVC in Visual Studio 2019 version 16.8**,” September 14, 2020. Accessed August 4, 2021. <https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/#private-module-fragments>.

- **Vassil Vassilev1, David Lange1, Malik Shahzad Muzaffar, Mircho Rodozov, Oksana Shadura and Alexander Penev**, “**C++ Modules in ROOT and Beyond**,” August 25, 2020. Accessed August 15, 2021. <https://arxiv.org/pdf/2004.06507.pdf>.
- **Bjarne Stroustrup**, “**Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules**,” June 12, 2020. Accessed August 14, 2021. <https://www.stroustrup.com/hop120main-p5-p-bfc9cd4--final.pdf>.
- **Rainer Grimm**, “**C++20: Further Open Questions to Modules**,” June 8, 2020. Accessed June 8, 2021. <https://www.modernescpp.com/index.php/c-20-open-questions-to-modules>.
- **Rainer Grimm**, “**C++20: Structure Modules**,” June 1, 2020. Accessed August 5, 2021. <https://www.modernescpp.com/index.php/c-20-divide-modules>.
- **Rainer Grimm**, “**C++20: Module Interface Unit and Module Implementation Unit**,” May 25, 2020. Accessed August 5, 2021. <https://www.modernescpp.com/index.php/c-20-module-interface-unit-and-module-implementation-unit>.
- **Rainer Grimm**, “**C++20: A Simple math Module**,” May 17, 2020. Accessed August 5, 2021. <https://www.modernescpp.com/index.php/cpp20-a-first-module>.
- **Corentin Jabot**, “**What do we want from a modularized Standard Library?**” May 16, 2020. Accessed August 12, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2172r0.pdf>
- **Rainer Grimm**, “**C++20: The Advantages of Modules**,” May 10, 2020. Accessed August 5, 2021. <https://www.modernescpp.com/index.php/cpp20-modules>.

- **Cameron DaCamara**, “**C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5**,” January 22, 2020. Accessed August 4, 2021.
<https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/>.
- **Colin Robertson and Nick Schonning**, “**Overview of modules in C++**,” December 13, 2019. Accessed August 13, 2021.
<https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160>.
- **Arthur O’Dwyer**, “**Hello World with C++2a modules**,” November 7, 2019. Accessed August 14, 2021.
<https://quuxplusone.github.io/blog/2019/11/07/module-hello-world/>.
- **Eric Niebler**, “**Understanding C++ Modules: Part 3: Linkage and Fragments**,” October 7, 2019. Accessed August 4, 2021.
<https://vector-of-bool.github.io/2019/10/07/modules-3.html>.
- **Rainer Grimm**, “**More Details to Modules**,” May 13, 2019. Accessed August 7, 2021. <http://modernescpp.com/index.php/c-20-more-details-to-modules>.
- **Rainer Grimm**, “**Modules**,” May 6, 2019. Accessed August 7, 2021. <http://modernescpp.com/index.php/c-20-modules>.
- **Bryce Adelstein Lelbach and Ben Craig**, “P1687R1: Summary of the Tooling Study Group’s Modules Ecosystem Technical Report Telecons,” August 5, 2019. Accessed August 18, 2021.
<https://lists.isocpp.org/sg15/attachment/0113/P1687R1.html>.
- **Corentin Jabot**, “**Naming guidelines for modules**,” June 16, 2019. Accessed August 18, 2021.
<https://isocpp.org/files/papers/P1634R0.html>.
- **Eric Niebler**, “**Understanding C++ Modules: Part 2: export, import, visible, and reachable**,” March 31, 2019. Accessed August 4, 2021.
<https://vector-of-bool.github.io/2019/03/31/modules-2.html>.

[bool.github.io/2019/03/31/modules-2.html](https://vector-of-bool.github.io/2019/03/31/modules-2.html).

- **Eric Niebler**, “**Understanding C++ Modules: Part 1: Hello Modules, and Module Units,**” March 10, 2019. Accessed August 4, 2021. <https://vector-of-bool.github.io/2019/03/10/modules-1.html>.
- **Rene Rivera**, “**Are modules fast? (revision 1)**,” March 6 2019, Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1441r1.pdf>
- **Richard Smith**, “**Merging Modules**,” February 22, 2019. Accessed August 13, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf>
- **Eric Niebler**, “**C++ Modules Might Be Dead-on-Arrival**,” January 27, 2019. Accessed August 18, 2021. <https://vector-of-bool.github.io/2019/01/27/modules-doa.html>.
- **Richard Smith and Gabriel Dos Reis**, “**Merging Modules.**” June 22, 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1103r0.pdf>
- **Gabriel Dos Reis and Richard Smith**, “**Modules for Standard C++**,” May 7 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1087r0.pdf>
- **Richard Smith**, “**Another take on Modules (Revision 1).**” March 6 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r1.pdf>
- **Bjarne Stroustrup**, “**Modules and macros**,” February 11, 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf>
- **Dmitry Guzeev**, “**A Few Words on C++ Modules**,” January 8, 2018. Accessed August 10, 2021. <https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64>.
- **Gabriel Dos Reis (Ed.)**, “**Working Draft, Extensions to C++ for Modules**,” January 29, 2018. Accessed August 15, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf>

[std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf).

- **Gabriel Dos Reis and Pavel Curtis, “Modules, Componentization, and Transition,”** October 5, 2015. Accessed August 6, 2021.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0141r0.pdf>
- **Gabriel Dos Reis, Mark Hall and Gor Nishanov, “A Module System for C++,”** May 27, 2014. Accessed August 15, 2021.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf>
- **Daveed Vandevoorde, “Modules in C++ (Revision 6),”** January 11, 2012.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf>.

Documentation

- **“3.23 C++ Modules.”** Accessed August 10, 2021.
https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.
- **“C++20 Standard: 10 Modules.”** Accessed August 12, 2021.
<https://eel.is/c++draft/module>.
- **“Modules—Module Partitions.”** Accessed August 13, 2021.
<https://en.cppreference.com/w/cpp/language/module>
- Wikipedia, **“Bjarne Stroustrup.”** Accessed August 14, 2021.
https://en.wikipedia.org/wiki/Bjarne_Stroustrup.

Appendix: Modules Glossary

- **export a declaration**—Make a declaration available to translation units that import the corresponding module.
- **export a definition**—Make a definition (such as a template) available to translation units that import the corresponding module.
- **export block**—A braced block of code preceded by `export`. A module exports all the declarations or definitions in the block. The block does not define a new scope.
- **export module declaration**—Indicates that a module unit is the

primary module interface unit and introduces the module's name.

- **global module**—An unnamed module that contains all identifiers defined in non-module translation units or in global module fragments.
- **global module fragment**—In a module unit, this fragment may contain only preprocessor directives and must appear before the module declaration. All declarations in this fragment are part of the global module and can be used throughout the remainder of the module unit.
- **header unit**—A header that is `imported` rather than `#included`.
- **IFC (.ifc) format**—A Microsoft Visual C++ file format for storing the information the compiler generates for a module.
- **import a header file**—Enables existing headers to be processed as header units, which can reduce compilation times in large projects.
- **import a module**—Make a module's `exported` declarations available in a translation unit.
- **import declaration**—A statement used to `import` a module into a translation unit.
- **interface dependency**—If you import a module into an implementation unit, the implementation unit has a dependency on that module's interface.
- **module**—
 - **module declaration**—Every module unit has a module declaration specifying the module's name and possibly a partition name.
 - **module implementation unit**—A module unit in which the module declaration does not begin with the `export` keyword.
 - **module interface unit**—A module unit in which the module declaration begins with the `export` keyword.
 - **module linkage**—Names that are in a module, but not exported from it are known only in that module.
 - **module name**—The name specified in a module's module declaration. All module units in a given module must have the same module name.
 - **module partition**—A module unit in which the module declaration

specifies the module name followed by a colon and a partition name. Module partition names in the same named module must be unique. If a module partition is a module interface partition it must be exported by the module's primary interface unit.

- **module purview**—The set of identifiers within a module unit from the `module` declaration to the end of the translation unit.
- **module unit**—An implementation unit containing a `module` declaration.
- **named module**—All module units with the same module name.
- **named module purview**—The purviews of all the module units in the named module.
- **namespace**—Defines a scope in which identifiers and variables are placed to help prevent naming conflicts with identifiers in your own programs and libraries.
- **partition**—A kind of module unit that defines a portion of a module's interface or implementation. Partitions are not visible to translation units that import the module.
- **precompiled module interface (.pcm)**—A `clang++` file that contains information about a module's interface. Used when compiling other translation units that depend on a given module.
- **primary module interface unit**—Determines the set of declarations exported by a module for use in other translation units.
- **:private module fragment**—A section in a primary module interface unit that enables you to define a module's implementation in the same file as its interface without exposing the implementation to other translation units.
- **reachable declaration**—A declaration is reachable if you can use it in your code without referencing it directly. For example, if you `import` a module into a translation unit and one of the module's exported functions returns an object of a non-exported type, that type is reachable in importing translation units.
- **tooling for modules**—Developer tools (under development) that will help developers implement and use modules.

- **transitive interface dependency**—import and re-export a module interface.
- **translation unit**—A preprocessed source-code file that is ready to be compiled.
- **visible declaration**—A declaration you can use by name in your code. For example, if you import a module into a translation unit, the module's exported declarations are visible in that translation unit.

Chapter 17. Concurrent Programming; Intro to C++20 Coroutines [This content is currently in development.]

This content is currently in development.

Part 6: Other Topics

Chapter 18. Stream I/O; C++20 Text Formatting: A Deeper Look [This content is currently in development.]

This content is currently in development.

Chapter 19. Other Topics; A Look Toward C++23 and Contracts [This content is currently in development.]

This content is currently in development.

Part 7: Appendices

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. Bits, Characters, C Strings and structs [This content is currently in development.]

This content is currently in development.

Appendix G. C Legacy Code Topics [This content is currently in development.]

This content is currently in development.

Appendix H. Using the Visual Studio Debugger [This content is currently in development.]

This content is currently in development.

Appendix I. Using the GNU C++ Debugger

[This content is currently in development.]

This content is currently in development.

Appendix J. 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

- 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>

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

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

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

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

```
~/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.
?
```

```
~/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.
?
```

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)?

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

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

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

```
#include <iostream> // enables program to output data to the screen
```

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

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

```
int number1{0}; // first integer to add (initialized to 0)
int number2{0}; // second integer to add (initialized to 0)
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

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

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

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

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

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

```
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

```
sum = number1 + number2; // add the numbers; store result in sum
```

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

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

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

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

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

```
sum = number1 + number2; // add the numbers; store result in sum
```

```
std::cout << "Sum is " << sum << std::endl; // display sum; end line
```

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

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

```
int number1{0}; // first integer to compare (initialized to 0)
int number2{0}; // second integer to compare (initialized to 0)
```

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

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

```
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++!

```
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++!

```
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\n to\n\nC++!\n";
8 } // end function main
```

Welcome
to

C++!

```
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
```

```
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

```
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
```

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

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

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

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

```
double average{static_cast<double>(total) / gradeCounter};
```

Type 'double' cannot be narrowed to 'int' in initializer list

conversion from 'double' to 'int' requires a narrowing conversion

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

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

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

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

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

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

```
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
```

```
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
```

```
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
```

```
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
```

```
1 // fig03_05.cpp
2 // Integer ranges and arbitrary precision integers.
3 #include <iostream>
4 #include "bignum.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 }
```

```
long long value1: 9223372036854775807  
value1 - 1: 9223372036854775806  
value1 + 1: -9223372036854775808
```

OK

Incorrect result

```
BigNumber value2: 9223372036854775807  
value2 - 1: 9223372036854775806  
value2 + 1: 9223372036854775808
```

OK

OK

```
value3: 100000000  
value3 to the power 2: 10000000000000000  
value3 to the power 3: 2003764205206896640  
value3 to the power 4: -8814407033341083648  
value3 to the power 5: -5047021154770878464
```

OK

Incorrect result

Incorrect result

Incorrect result

```
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

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

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

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

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

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

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

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

```
switch (controllingExpression) {  
    case 7:  
        // statements  
        break;  
    [[likely]] case 11:  
        // statements  
        break;  
    default:  
        // statements  
        break;  
}
```

```
8 int main() {  
9     cout << "Enter a ZIP file name: ";  
10    string zipFileName;  
11    getline(cin, zipFileName); // inputs a line of text  
12
```

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

```
13 // strings literals separated only by whitespace are combined
14 // into a single string by the compiler
15 string content{
16     "This chapter introduces all but one of the remaining control "
17     "statements--the for, do...while, switch, break and continue "
18     "statements. We explore the essentials of counter-controlled "
19     "iteration. We use compound-interest calculations to begin "
20     "investigating the issues of processing monetary amounts. First, "
21     "we discuss the representational errors associated with "
22     "floating-point types. We use a switch statement to count the "
23     "number of A, B, C, D and F grade equivalents in a set of "
24     "numeric grades. We show C++17's enhancements that allow you to "
25     "initialize one or more variables of the same type in the "
26     "headers of if and switch statements."};
27
```

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

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

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

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

```
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

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

```
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
```

```
50 // original file contents
51 string extractedContent{input.read(info)};
52
```

```
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--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.

```
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
```

```
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
```

```
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

```
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

```
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
```

```
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             // 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

```
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
```

```
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

```
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
```

```
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

```
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
```

```
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
```

```
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: {:>7.2f}\n", principal)
14         << format("    Interest rate: {:>7.2f}\n", rate);
15 }
```

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

```
Initial principal: 1000.00
Interest rate:    0.05
```

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

```
int maximum(int x, int y, int z); // function prototype
```

6 6 5 5 6 5 1 1 5 3

```
    srand(gsl::narrow_cast<unsigned int>(time(0)));
```

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

```
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
```

```
21 // determine game status and point (if needed) based on first roll
22 switch (const int sumOfDice{rollDice()}; sumOfDice) {
23     case 7: // win with 7 on first roll
24     case 11: // win with 11 on first roll
25         gameStatus = Status::won;
26         break;
27     case 2: // lose with 2 on first roll
28     case 3: // lose with 3 on first roll
29     case 12: // lose with 12 on first roll
30         gameStatus = Status::lost;
31         break;
32     default: // did not win or lose, so remember point
33         myPoint = sumOfDice; // remember the point
34         cout << "Point is " << myPoint << endl;
35         break; // optional at end of switch
36     }
37
```

```
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()}; sumOfDice == myPoint) {
42         gameStatus = Status::won;
43     }
44     else {
45         if (sumOfDice == 7) { // lose by rolling 7 before point
46             gameStatus = Status::lost;
47         }
48     }
49 }
50
```

```
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
```

```
60 // roll dice, calculate sum and display results
61 int rollDice() {
62     const int die1{1 + rand() % 6}; // first die roll
63     const int die2{1 + rand() % 6}; // second die roll
64     const int sum{die1 + die2}; // compute sum of die values
65
66     // display results of this roll
```

```
67     cout << "Player rolled " << die1 << " + " << die2  
68     << " = " << sum << endl;  
69     return sum;  
70 }
```

```
enum class Months {jan = 1, feb, mar, apr, may, jun, jul, aug,  
    sep, oct, nov, dec};
```

```
enum Status {keepRolling, won, lost};
```

```
enum class Status : long {keepRolling, won, lost};
```

```
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
```

```
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
```

```
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 }
```

```
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
```

```
59 // useGlobal modifies global variable x during each call
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 }
```

```
int count{1}; // declare integer variable count
int& cRef{count}; // create cRef as an alias for count
++cRef; // increment count (using its alias cRef)
```

```
void displayName(std::string name) {  
    std::cout << name << std::endl;  
}
```

```
int maximum(int value1, int value2, int value3) {  
    int maximumValue=value1; // assume value1 is maximum  
    // determine whether value2 is greater than maximumValue  
    if (value2 > maximumValue) {  
        maximumValue = value2;  
    }  
    // determine whether value3 is greater than maximumValue  
    if (value3 > maximumValue) {  
        maximumValue = value3;  
    }  
    return maximumValue;  
}
```

```
int factorial{1};  
for (int counter{number}; counter >= 1; --counter) {  
    factorial *= counter;  
}
```

```
if (s.empty()) {  
    // do something because the string s is empty  
}
```

[[nodiscard("Insight into why return value should not be ignored")]]

```
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
The maximum integer value is: 86
```

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

```
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

```
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

```
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

```
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 "gsl/gsl" // Guidelines Support Library for narrow_cast
7 using namespace std;
8
9 int rollDice(); // rolls dice, calculates and displays sum
10
11 int main() {
```

```
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 "gsl/gsl"
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{gsl::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
```

```
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
```

```
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
```

```
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
```

```
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

```
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
```

```
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
```

```
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
_Z8nothinglifcRi
_Z8nothing2ciRfRd
main
```

```
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 }
```

```
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

```
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
```

```
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
```

```
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
```

```
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 }
```

```
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 JnaqnL++ bknsfbxfew eztlinmyahc xdro Z++20!

Decrypted:
Welcome to Modern C++ application development with C++20!

Enter the ciphertext to decipher:
Lnfylun Lhqtmoh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqh1z

Decrypted:
Objects Natural Case Study: Encryption and Decryption

```
array<int, 5> c; // c is an array of 5 int values
```

terminate called after throwing an instance of 'std::out_of_range'
what(): array::at: __n (which is 10) >= _Nm (which is 5)
Aborted

```
array<int, 5> n{32, 27, 64, 18, 95, 14};
```

```
array<int, 5> n{}; // initialize elements of array n to 0
```

```
for (int counter{0}; counter < items.size(); ++counter) {  
    cout << items[counter] << " ";  
}
```

```
for (size_t row{0}; row < a.size(); ++row) {
    for (size_t column{0}; column < a.at(row).size(); ++column) {
        cout << a.at(row).at(column) << ' '; // a[row][column]
    }
    cout << endl;
}
```

```
const array<array<int, columns>, rows> array1{  
    {{1, 2, 3}, // row 0  
     {4, 5, 6}} // row 1  
};
```

```
for (size_t column{0}; column < 4; ++column) {  
    a.at(2).at(column) = 0; // a[2][column]  
}
```

```
a.at(2)(0) = 0; // a[2][0]
a.at(2)(1) = 0; // a[2][1]
a.at(2)(2) = 0; // a[2][2]
a.at(2)(3) = 0; // a[2][3]
```

```
total = 0;

for (size_t row{0}; row < a.size(); ++row) {
    for (size_t column{0}; column < a.at(row).size(); ++column) {
        total += a.at(row).at(column); // a[row][column]
    }
}
```

```
total = 0;  
for (auto row : a) { // for each row  
    for (auto column : row) { // for each column in row  
        total += column;  
    }  
}
```

```
for (const int item : integers) {  
    total += item;  
}
```

```
[](const auto& x, const auto& y){return x * y;}
```

```
21 auto values1 = views::iota(1, 11); // generate integers 1-10
22 showValues(values1, "Generate integers 1-10");
23
```

Generate integers 1-10: 1 2 3 4 5 6 7 8 9 10

```
values1 | views::filter([](const auto& x) {return x % 2 == 0;});
```

```
24 // filter each value in values1, keeping only the even integers
25 auto values2 =
26     values1 | views::filter([](const auto& x) {return x % 2 == 0;});
27 showValues(values2, "Filtering even integers");
28
```

Filtering even integers: 2 4 6 8 10

```
29 // map each value in values2 to its square
30 auto values3 =
31     values2 | views::transform([](const auto& x) {return x * x;});
32 showValues(values3, "Mapping even integers to squares");
33
```

Mapping even integers to squares: 4 16 36 64 100

```
34 // combine filter and transform to get squares of the even integers
35 auto values4 =
36     values1 | views::filter([](const auto& x) {return x % 2 == 0;})
37         | views::transform([](const auto& x) {return x * x;});
38 showValues(values4, "Squares of even integers");
39
```

Squares of even integers: 4 16 36 64 100

```
40 // total the squares of the even integers
41 cout << "Sum the squares of the even integers from 2-10: " <<
42     accumulate(begin(values4), end(values4), 0) << endl;
43
```

Sum the squares of the even integers from 2-10: 220

```
44 // process a container's elements
45 array<int, 10> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
46 auto values5 =
47     numbers | views::filter([](const auto& x) {return x % 2 == 0;})
48         | views::transform([](const auto& x) {return x * x;});
49 showValues(values5, "Squares of even integers in array numbers");
50 }
```

Squares of even integers in array numbers: 4 16 36 64 100

```
12 int main() {  
13     vector<int> integers1(7); // 7-element vector<int>  
14     vector<int> integers2(10); // 10-element vector<int>  
15 }
```

```
16 // print integers1 size and contents
17 cout << "Size of vector integers1 is " << integers1.size()
18     << "\nvector after initialization:";
19 outputVector(integers1);
20
21 // print integers2 size and contents
22 cout << "\nSize of vector integers2 is " << integers2.size()
23     << "\nvector after initialization:";
24 outputVector(integers2);
25
```

```
Size of vector integers1 is 7
vector after initialization: 0 0 0 0 0 0 0
```

```
Size of vector integers2 is 10
vector after initialization: 0 0 0 0 0 0 0 0 0 0
```

```
26 // input and print integers1 and integers2
27 cout << "\nEnter 17 integers:" << endl;
28 inputVector(integers1);
29 inputVector(integers2);
30
31 cout << "\nAfter input, the vectors contain:\n"
32     << "integers1:";
33 outputVector(integers1);
34 cout << "integers2:";
35 outputVector(integers2);
36
```

```
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

```
After input, the vectors contain:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17
```

```
37 // use inequality (!=) operator with vector objects
38 cout << "\nEvaluating: integers1 != integers2" << endl;
39
40 if (integers1 != integers2) {
41     cout << "integers1 and integers2 are not equal" << endl;
42 }
43
```

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

```
44 // create vector integers3 using integers1 as an
45 // initializer; print size and contents
46 vector<int> integers3{integers1}; // copy constructor
47
48 cout << "\nSize of vector integers3 is " << integers3.size()
49     << "\nvector after initialization: ";
50 outputVector(integers3);
51
```

Size of vector integers3 is 7
vector after initialization: 1 2 3 4 5 6 7

```
52 // use overloaded assignment (=) operator
53 cout << "\nAssigning integers2 to integers1:" << endl;
54 integers1 = integers2; // assign integers2 to integers1
55
56 cout << "integers1: ";
57 outputVector(integers1);
58 cout << "integers2: ";
59 outputVector(integers2);
60
61 // use equality (==) operator with vector objects
62 cout << "\nEvaluating: integers1 == integers2" << endl;
63
64 if (integers1 == integers2) {
65     cout << "integers1 and integers2 are equal" << endl;
66 }
67
```

Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

```
68 // use square brackets to use the value at location 5 as an rvalue
69 cout << "\nintegers1.at(5) is " << integers1.at(5);
70
71 // use square brackets to create lvalue
72 cout << "\n\nAssigning 1000 to integers1.at(5)" << endl;
73 integers1.at(5) = 1000;
74 cout << "integers1: ";
75 outputVector(integers1);
76
```

```
integers1.at(5) is 13
Assigning 1000 to integers1.at(5)
integers1: 8 9 10 11 12 1000 14 15 16 17
```

```
77 // attempt to use out-of-range index
78 try {
79     cout << "\nAttempt to display integers1.at(15)" << endl;
80     cout << integers1.at(15) << endl; // ERROR: out of range
81 }
82 catch (const out_of_range& ex) {
83     cerr << "An exception occurred: " << ex.what() << endl;
84 }
85
```

```
Attempt to display integers1.at(15)
An exception occurred: invalid vector<T> subscript
```

```
86 // changing the size of a vector
87 cout << "\nCurrent integers3 size is: " << integers3.size() << endl;
88 integers3.push_back(1000); // add 1000 to the end of the vector
89 cout << "New integers3 size is: " << integers3.size() << endl;
90 cout << "integers3 now contains: ";
91 outputVector(integers3);
92 }
93
```

Current integers3 size is: 7
New integers3 size is: 8
integers3 now contains: 1 2 3 4 5 6 7 1000

```
94 // output vector contents
95 void outputVector(const vector<int>& items) {
96     for (const int item : items) {
97         cout << item << " ";
98     }
99
100    cout << endl;
101 }
102
103 // input vector contents
104 void inputVector(vector<int>& items) {
105     for (int& item : items) {
106         cin >> item;
107     }
108 }
```

```

1 // fig06_01.cpp
2 // Initializing an array's elements to zeros and printing the array.
3 #include "fmt/format.h" // C++20: This will be #include <format>
4 #include <iostream>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     array<int, 5> n; // n is an array of 5 int values
10
11    // initialize elements of array n to 0
12    for (size_t i{0}; i < n.size(); ++i) {
13        n[i] = 0; // set element at location i to 0
14    }
15
16    cout << fmt::format("{}{:>10}\n", "Element", "Value");
17
18    // output each array element's value
19    for (size_t i{0}; i < n.size(); ++i) {
20        cout << fmt::format("{:>7}{:>10}\n", i, n[i]);
21    }
22
23    cout << "\nElement" << setw(10) << "Value" << endl;
24
25    // access elements via the at member function
26    for (size_t i{0}; i < n.size(); ++i) {
27        cout << fmt::format("{:>7}{:>10}\n", i, n.at(i));
28    }
29
30    // accessing an element outside the array's bounds with at
31    cout << n.at(10) << endl;
32}

```

Element	Value
0	0
1	0
2	0
3	0
4	0

Element	Value
0	0
1	0
2	0
3	0
4	0

terminate called after throwing an instance of 'std::out_of_range'
what(): array::at: __n (which is 10) >= _Nm (which is 5)
Aborted

```
1 // fig06_02.cpp
2 // Initializing an array in a declaration.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     array<int, 5> n{32, 27, 64, 18, 95}; // list initializer
10
11    // output each array element's value
12    for (size_t i{0}; i < n.size(); ++i) {
13        cout << n.at(i) << " ";
14    }
15
16    cout << endl;
17 }
```

```
32 27 64 18 95
```

```
1 // fig06_03.cpp
2 // Using range-based for.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     array<int, 5> items{1, 2, 3, 4, 5}; // list initializer
9
10    // display items before modification
11    cout << "items before modification: ";
12    for (const int item : items) {
13        cout << item << " ";
14    }
15
16    // multiply the elements of items by 2
17    for (int& itemRef : items) { // itemRef is a reference to an int
18        itemRef *= 2;
19    }
20
21    // display items after modification
22    cout << "\nitems after modification: ";
23    for (const int item : items) {
24        cout << item << " ";
25    }
26
27    // total elements of items using range-based for with initialization
28    cout << "\n\ncalculating a running total of items' values: ";
29    for (int runningTotal{0}; const int item : items) {
30        runningTotal += item;
31        cout << "\nitem: " << item << "; running total: " << runningTotal;
32    }
33
34    cout << endl;
35 }
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10
calculating a running total of items' values:
item: 2; running total: 2
item: 4; running total: 6
item: 6; running total: 12
item: 8; running total: 20
item: 10; running total: 30
```

```
1 // fig06_04.cpp
2 // Set array s to the even integers from 2 to 10.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     // constant can be used to specify array size
9     constexpr size_t arraySize{5}; // must initialize in declaration
10
11     array<int, arraySize> values{}; // array values has 5 elements
12
13     for (size_t i{0}; i < values.size(); ++i) { // set the values
14         values.at(i) = 2 + 2 * i;
15     }
16
17     // output contents of array s in tabular format
18     for (const int value : values) {
19         cout << value << " ";
20     }
21
22     cout << endl;
23 }
```

```
2 4 6 8 10
```

```
1 // fig06_05.cpp
2 // Compute the sum of the elements of an array.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     constexpr size_t arraySize{4};
9     array<int, arraySize> integers{10, 20, 30, 40};
10    int total{0};
11
12    // sum contents of array a
13    for (const int item : integers) {
14        total += item;
15    }
16
17    cout << "Total of array elements: " << total << endl;
18 }
```

Total of array elements: 100

```
1 // fig06_06.cpp
2 // Printing a student grade distribution as a primitive bar chart.
3 #include <iostream>
4 #include <array>
```

```
5  using namespace std;
6
7  int main() {
8      constexpr size_t arraySize{11};
9      array<int, arraySize> n{0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
10
11     cout << "Grade distribution:" << endl;
12
13     // for each element of array n, output a bar of the chart
14     for (int i{0}; const int item : n) {
15         // output bar labels ("0-9:", ..., "90-99:", "100:")
16         if (0 == i) {
17             cout << " 0-9: ";
18         }
19         else if (10 == i) {
20             cout << " 100: ";
21         }
22         else {
23             cout << i * 10 << "-" << (i * 10) + 9 << ": ";
24         }
25
26         ++i;
27
28         // print bar of asterisks
29         for (int stars{0}; stars < item; ++stars) {
30             cout << '*';
31         }
32
33         cout << endl; // start a new line of output
34     }
35 }
```

Grade distribution:

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ***
90-99: **
100: *

```
1 // fig06_07.cpp
2 // Die-rolling program using an array instead of switch.
3 #include "fmt/format.h" // C++20: This will be #include <format>
4 #include <iostream>
5 #include <array>
6 #include <random>
7 #include <ctime>
8 #include "gsl/gsl"
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(gsl::narrow_cast<unsigned int>(time(0)));
15     uniform_int_distribution<int> randomInt(1, 6);
16
17     constexpr size_t arraySize{7}; // ignore element zero
18     array<int, arraySize> frequency{}; // initialize to 0s
19
20     // roll die 60,000,000 times; use die value as frequency index
21     for (int roll{1}; roll <= 60'000'000; ++roll) {
22         ++frequency.at(randomInt(engine));
23     }
24
25     cout << fmt::format("[]{:>13}\n", "Face", "Frequency");
26
27     // output each array element's value
28     for (size_t face{1}; face < frequency.size(); ++face) {
29         cout << fmt::format("{:>4} {:>13}\n", face, frequency.at(face));
30     }
31 }
```

Face	Frequency
1	9997901
2	9999110
3	10001172
4	10003619
5	9997606
6	10000592

```
1 // fig06_08.cpp
2 // Poll analysis program.
3 #include "fmt/format.h" // C++20: This will be #include <format>
4 #include <iostream>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     // define array sizes
10    constexpr size_t responseSize{20}; // size of array responses
11    constexpr size_t frequencySize{6}; // size of array frequency
12
13    // place survey responses in array responses
14    const array<int, responseSize> responses{
15        1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
16
17    // initialize frequency counters to 0
18    array<int, frequencySize> frequency{};
19
20    // for each answer, select responses element and use that value
21    // as frequency index to determine element to increment
22    for (size_t answer{0}; answer < responses.size(); ++answer) {
23        ++frequency.at(responses.at(answer));
24    }
25
26    cout << fmt::format("{}{:>12}\n", "Rating", "Frequency");
27
28    // output each array element's value
29    for (size_t rating{1}; rating < frequency.size(); ++rating) {
30        cout << fmt::format("{:>6}{:>12}\n", rating, frequency.at(rating));
31    }
32 }
```

Rating	Frequency
1	3
2	5
3	7
4	2
5	3

```
1 // fig06_09.cpp
2 // Sorting and searching arrays.
3 #include <iostream>
4 #include <array>
5 #include <string>
6 #include <algorithm> // contains sort and binary_search
7 using namespace std;
8
9 int main() {
10     constexpr size_t arraySize{7}; // size of array colors
11     array<string, arraySize> colors{"red", "orange", "yellow",
12         "green", "blue", "indigo", "violet"};
13
14     // output original array
15     cout << "Unsorted colors array:\n";
16     for (string color : colors) {
17         cout << color << " ";
18     }
19
20     sort(begin(colors), end(colors)); // sort contents of colors
21
22     // output sorted array
23     cout << "\nSorted colors array:\n";
24     for (string item : colors) {
25         cout << item << " ";
26     }
27
28     // search for "indigo" in colors
29     bool found{binary_search(begin(colors), end(colors), "indigo")};
30     cout << "\n\n\"indigo\" " << (found ? "was" : "was not")
31         << " found in colors array" << endl;
32 }
```

```
33 // search for "cyan" in colors
34 found = binary_search(begin(colors), end(colors), "cyan");
35 cout << "\"cyan\" " << (found ? "was" : "was not")
36     << " found in colors array" << endl;
37 }
```

Unsorted colors array:

red orange yellow green blue indigo violet

Sorted colors array:

blue green indigo orange red violet yellow

"indigo" was found in colors array

"cyan" was not found in colors array

```
1 // fig06_10.cpp
2 // Initializing multidimensional arrays.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 constexpr size_t rows{2};
8 constexpr size_t columns{3};
9 void printArray(const array<array<int, columns>, rows>& a);
10
11 int main() {
12     const array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};
13     const array<array<int, columns>, rows> array2{1, 2, 3, 4, 5};
14
15     cout << "Values in array1 by row are:" << endl;
16     printArray(array1);
17
18     cout << "\nValues in array2 by row are:" << endl;
19     printArray(array2);
20 }
21
22 // output array with two rows and three columns
23 void printArray(const array<array<int, columns>, rows>& a) {
24     // loop through array's rows
25     for (auto const& row : a) {
26         // loop through columns of current row
27         for (auto const& element : row) {
28             cout << element << ' ';
29         }
30
31         cout << endl; // start new line of output
32     }
33 }
```

Values in array1 by row are:

1 2 3
4 5 6

Values in array2 by row are:

1 2 3
4 5 0

```
1 // fig06_11.cpp
2 // Compute the sum of the elements of an array.
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 using namespace std;
7
8 int main() {
9     constexpr size_t arraySize{4};
10    array<int, arraySize> integers{10, 20, 30, 40};
11    cout << "Total of array elements: " <<
12        accumulate(begin(integers), end(integers), 0) << endl;
13 }
```

Total of array elements: 100

```
1 // fig06_12.cpp
2 // Compute the product of an array's elements using accumulate.
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 using namespace std;
7
8 int multiply(int x, int y) {
9     return x * y;
10 }
11
12 int main() {
13     constexpr size_t arraySize{5};
14     array<int, arraySize> integers{1, 2, 3, 4, 5};
15
16     cout << "Product of integers: "
17         << accumulate(begin(integers), end(integers), 1, multiply) << endl;
18
19     cout << "Product of integers with a lambda: "
20         << accumulate(begin(integers), end(integers), 1,
21                         [] (const auto& x, const auto& y){return x * y;}) << endl;
22 }
```

Product of integers: 120

Product of integers with a lambda: 120

```
1 // fig06_13.cpp
2 // Functional-style programming with C++20 ranges and views.
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 #include <ranges>
7 using namespace std;
8
9 int main() {
10     // Lambda to display results of range operations
11     auto showValues = [] (auto& values, const string& message) {
12         cout << message << ": ";
13
14         for (auto value : values) {
15             cout << value << " ";
16         }
17
18         cout << endl;
19     };
20 }
```

```
1 // fig06_14.cpp
2 // Demonstrating C++ standard library class template vector.
3 #include <iostream>
4 #include "fmt/format.h" // C++20: This will be #include <format>
5 #include <vector>
6 #include <stdexcept>
7 using namespace std;
8
9 void outputVector(const vector<int>& items); // display the vector
10 void inputVector(vector<int>& items); // input values into the vector
11
```

```
int y{5}; // declare variable y
int* yPtr=nullptr; // declare pointer variable yPtr
```

```
yPtr = &y; // assign address of y to yPtr
```

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube *nPtr
```

```
int c[5]; // c is a built-in array of 5 integers
```

```
int sumElements(const int values[], size_t numberOfElements)
```

```
int sumElements(const int* values, size_t numberOfElements)
```

```
sort(begin(colors), end(colors)); // sort contents of colors
```

```
sort(begin(n), end(n)); // sort contents of built-in array n
```

```
15 const int values1[3]{10, 20, 30};  
16  
17 // creating a std::array from a built-in array  
18 const auto array1 = to_array(values1);  
19  
20 cout << "array1.size() = " << array1.size() << "\narray1: ";  
21 display(array4); // use lambda to display contents  
22
```

```
array1.size() = 3  
array1: 10 20 30
```

```
23 // creating a std::array from an initializer list
24 const auto array2 = to_array({1, 2, 3, 4});
25 cout << "\n\narray2.size() = " << array2.size() << "\narray2: ";
26 display(array2); // use lambda to display contents
27
28 cout << endl;
29
```

```
array2.size() = 4
array2: 1 2 3 4
```

```
10 // items parameter is treated as a const int* so we also need the size to
11 // know how to iterate over items with counter-controlled iteration
12 void displayArray(const int items[], size_t size) {
13     for (size_t i{0}; i < size; ++i) {
14         cout << items[i] << " ";
15     }
16 }
17
```

```
18 // span parameter contains both the location of the first item
19 // and the number of elements, so we can iterate using range-based for
20 void displaySpan(span<const int> items) {
21     for (const auto& item : items) { // spans are iterable
22         cout << item << " ";
23     }
24 }
25
```

```
26 // spans can be used to modify elements in the original data structure
27 void times2(span<int> items) {
28     for (int& item : items) {
29         item *= 2;
30     }
31 }
32
```

```
33 int main() {
34     int values1[5]{1, 2, 3, 4, 5};
35     array<int, 5> values2{6, 7, 8, 9, 10};
36     vector<int> values3{11, 12, 13, 14, 15};
37
38     // must specify size because the compiler treats displayArray's items
39     // parameter as a pointer to the first element of the argument
40     cout << "values1 via displayArray: ";
41     displayArray(values1, 5);
42 }
```

```
values1 via displayArray: 1 2 3 4 5
```

```
43 // compiler knows values' size and automatically creates a span
44 // representing &values1[0] and the array's length
45 cout << "\nvalues1 via displaySpan: ";
46 displaySpan(values1);
47
48 // compiler also can create spans from std::arrays and std::vectors
49 cout << "\nvalues2 via displaySpan: ";
50 displaySpan(values2);
51 cout << "\nvalues3 via displaySpan: ";
52 displaySpan(values3);
53
```

```
values1 via displayArray: 1 2 3 4 5
values1 via displaySpan: 1 2 3 4 5
values2 via displaySpan: 6 7 8 9 10
values3 via displaySpan: 11 12 13 14 15
```

```
54 // changing a span's contents modifies the original data
55 times2(values1);
56 cout << "\n\nvalues1 after times2 modifies its span argument: ";
57 displaySpan(values1);
58
```

```
values1 after times2 modifies its span argument: 2 4 6 8 10
```

```
59 // spans have various array-and-vector-like capabilities
60 span<int> mySpan{values1};
61 cout << "\n\nmySpan's first element: " << mySpan.front()
62     << "\nmySpan's last element: " << mySpan.back();
63
```

```
mySpan's first element: 2
mySpan's last element: 10
```

```
64 // spans can be used with standard library algorithms
65 cout << "\n\nSum of mySpan's elements: "
66     << accumulate(begin(mySpan), end(mySpan), 0);
67
```

Sum of mySpan's elements: 30

```
68 // spans can be used to create subviews of a container
69 cout << "\n\nFirst three elements of mySpan: ";
70 displaySpan(mySpan.first(3));
71 cout << "\nLast three elements of mySpan: ";
72 displaySpan(mySpan.last(3));
73 cout << "\nMiddle three elements of mySpan: ";
74 displaySpan(mySpan.subspan(1, 3));
75
```

First three elements of mySpan: 2 4 6

Last three elements of mySpan: 6 8 10

Middle three elements of mySpan: 4 6 8

```
76 // changing a subview's contents modifies the original data
77 times2(mySpan.subspan(1, 3));
78 cout << "\n\nvalues1 after modifying middle three elements via span: ";
79 displaySpan(values1);
80
```

```
values1 after modifying middle three elements via span: 2 8 12 16 10
```

```
81 // access a span element via []
82 cout << "\n\nThe element at index 2 is: " << mySpan[2];
83
84 // attempt to access an element outside the bounds
85 cout << "\n\nThe element at index 10 is: " << mySpan[10] << endl;
86 }
```

The element at index 2 is: 12

```
char color[]{'b', 'l', 'u', 'e', '\0'};
```

```
15 // initializing an array with a string literal
16 // creates a one-element array<const char*>
17 const auto array1 = array{"abc"};
18 cout << "\n\narray1.size() = " << array1.size() << "\narray1: ";
19 display(array1); // use lambda to display contents
20
```

```
array1.size() = 1
array1: abc
```

```
21 // creating std::array of characters from a string literal
22 const auto array2 = to_array("C++20");
23 cout << "\n\narray2.size() = " << array2.size() << "\narray2: ";
24 display(array2); // use lambda to display contents
25
26 cout << endl;
27 }
```

```
array2.size() = 6
array2: C + + 2 0
```

```
1 // fig07_01.cpp
2 // Pointer operators & and *.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     constexpr int a{7}; // initialize a with 7
8     const int* aPtr = &a; // initialize aPtr with address of int variable a
9
10    cout << "The address of a is " << &a
11        << "\nThe value of aPtr is " << aPtr;
12    cout << "\n\nThe value of a is " << a
13        << "\nThe value of *aPtr is " << *aPtr << endl;
14 }
```

```
The address of a is 002DFD80
The value of aPtr is 002DFD80
```

```
The value of a is 7
The value of *aPtr is 7
```

```
1 // fig07_02.cpp
2 // Pass-by-value used to cube a variable's value.
3 #include <iostream>
4 using namespace std;
5
6 int cubeByValue(int n); // prototype
7
8 int main() {
9     int number{5};
10
11    cout << "The original value of number is " << number;
12    number = cubeByValue(number); // pass number by value to cubeByValue
13    cout << "\nThe new value of number is " << number << endl;
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
18     return n * n * n; // cube local variable n and return result
19 }
```

The original value of number is 5
The new value of number is 125

```
1 // fig07_03.cpp
2 // Pass-by-reference with a pointer argument used to cube a
3 // variable's value.
4 #include <iostream>
5 using namespace std;
6
7 void cubeByReference(int* nPtr); // prototype
8
9 int main() {
10     int number{5};
11
12     cout << "The original value of number is " << number;
13     cubeByReference(&number); // pass number address to cubeByReference
14     cout << "\nThe new value of number is " << number << endl;
15 }
16
17 // calculate cube of *nPtr; modifies variable number in main
18 void cubeByReference(int* nPtr) {
19     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
20 }
```

The original value of number is 5
The new value of number is 125

```
1 // fig07_06.cpp
2 // C++20: Creating std::arrays with to_array.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     // lambda to display a collection of items
9     const auto display = [] (const auto& items) {
10         for (const auto& item : items) {
11             cout << item << " ";
12         }
13     };
14 }
```

20

```
1 // fig07_07.cpp
2 // Attempting to modify data through a
3 // nonconstant pointer to constant data.
4
5 int main() {
6     int y{0};
7     const int* yPtr{&y};
8     *yPtr = 100; // error: cannot modify a const object
9 }
```

GNU C++ compiler error message:

```
fig07_07.cpp: In function 'int main()':
fig07_07.cpp:8:10: error: assignment of read-only location '* yPtr'
8 |     *yPtr = 100; // error: cannot modify a const object
|~~~~~^~~~~~
```

```
1 // fig07_08.cpp
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11    *ptr = 7; // allowed: *ptr is not const
12    ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

Microsoft Visual C++ compiler error message:

```
error C3892: 'ptr': you cannot assign to a variable that is const
```

```
1 // fig07_09.cpp
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int x{5}, y;
8
9     // ptr is a constant pointer to a constant integer.
10    // ptr always points to the same location; the integer
11    // at that location cannot be modified.
12    const int* const ptr{&x};
13
14    cout << *ptr << endl;
15
16    *ptr = 7; // error: *ptr is const; cannot assign new value
17    ptr = &y; // error: ptr is const; cannot assign new address
18 }
```

Apple Clang compiler error messages:

```
fig07_09.cpp:16:9: error: read-only variable is not assignable
    *ptr = 7; // error: *ptr is const; cannot assign new value
    ~~~~ ^
fig07_09.cpp:17:8: error: cannot assign to variable 'ptr' with const-qualified type 'const int *const'
    ptr = &y; // error: ptr is const; cannot assign new address
    ~~~~ ^
```

```
1 // fig07_10.cpp
2 // Sizeof operator when used on a built-in array's name
3 // returns the number of bytes in the built-in array.
4 #include <iostream>
5 using namespace std;
6
7 size_t getSize(double* ptr); // prototype
8
9 int main() {
10    double numbers[20]; // 20 doubles; occupies 160 bytes on our system
11
12    cout << "The number of bytes in the array is " << sizeof(numbers);
13
14    cout << "\nThe number of bytes returned by getSize is "
15        << getSize(numbers) << endl;
16 }
17
18 // return size of ptr
19 size_t getSize(double* ptr) {
20    return sizeof(ptr);
21 }
```

The number of bytes in the array is 160
The number of bytes returned by getSize is 4

```
1 // fig07_11.cpp
2 // sizeof operator used to determine standard data type sizes.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    constexpr char c{}; // variable of type char
8    constexpr short s{}; // variable of type short
```

```

9  constexpr int i{}; // variable of type int
10 constexpr long l{}; // variable of type long
11 constexpr long long ll{}; // variable of type long long
12 constexpr float f{}; // variable of type float
13 constexpr double d{}; // variable of type double
14 constexpr long double ld{}; // variable of type long double
15 constexpr int array[20]{}; // built-in array of int
16 const int* const ptr{array}; // variable of type int*
17
18 cout << "sizeof c = " << sizeof c
19     << "\nsizeof(char) = " << sizeof(char)
20     << "\nsizeof s = " << sizeof s
21     << "\nsizeof(short) = " << sizeof(short)
22     << "\nsizeof i = " << sizeof i
23     << "\nsizeof(int) = " << sizeof(int)
24     << "\nsizeof l = " << sizeof l
25     << "\nsizeof(long) = " << sizeof(long)
26     << "\nsizeof ll = " << sizeof ll
27     << "\nsizeof(long long) = " << sizeof(long long)
28     << "\nsizeof f = " << sizeof f
29     << "\nsizeof(float) = " << sizeof(float)
30     << "\nsizeof d = " << sizeof d
31     << "\nsizeof(double) = " << sizeof(double)
32     << "\nsizeof ld = " << sizeof ld
33     << "\nsizeof(long double) = " << sizeof(long double)
34     << "\nsizeof array = " << sizeof array
35     << "\nsizeof ptr = " << sizeof ptr << endl;
36 }

```

sizeof c = 1	sizeof(char) = 1
sizeof s = 2	sizeof(short) = 2
sizeof i = 4	sizeof(int) = 4
sizeof l = 8	sizeof(long) = 8
sizeof ll = 8	sizeof(long long) = 8
sizeof f = 4	sizeof(float) = 4
sizeof d = 8	sizeof(double) = 8
sizeof ld = 16	sizeof(long double) = 16
sizeof array = 80	
sizeof ptr = 8	

```
1 // fig07_12.cpp
2 // C++20 spans: Creating views into containers.
3 #include <array>
4 #include <iostream>
5 #include <numeric>
6 #include <span>
7 #include <vector>
8 using namespace std;
9
```

```
1 // fig07_13.cpp
2 // Reading in command-line arguments.
3 #include <iostream>
4 using namespace std;
5
6 int main(int argc, char* argv[]) {
7     cout << "There were " << argc << " command-line arguments:\n";
8     for (int i{0}; i < argc; ++i) {
9         cout << argv[i] << endl;
10    }
11 }
```

fig07_13 Amanda Green 97

There were 4 command-line arguments

fig07_13
Amanda
Green
97

```
1 // fig07_14.cpp
2 // C++20: Creating std::arrays from string literals with to_array.
3 #include <iostream>
4 #include <array>
5 using namespace std;
6
7 int main() {
8     // lambda to display a collection of items
9     const auto display = [](const auto& items) {
10         for (const auto& item : items) {
11             cout << item << " ";
12         }
13     };
14 }
```

```
string bar{8, '*'}; // string of 8 '*' characters
```

```
9  string s1{"cat"};
10 string s2; // initialized to the empty string
11 string s3; // initialized to the empty string
12
13 s2 = s1; // assign s1 to s2
14 s3.assign(s1); // assign s1 to s3
15 cout << fmt::format("s1: {}\ns2: {}\ns3: {}", s1, s2, s3);
16
```

s1: cat
s2: cat
s3: cat

```
target.assign(source, start, numberOfChars);
```

```
17     s2.at(0) = 'r'; // modify s2
18     s3.at(2) = 'r'; // modify s3
19     cout << fmt::format("After modifications:\ns2: {}\n\ns3: {}", s2, s3);
20
```

After modifications:

s2: rat

s3: car

```
21 cout << "\n\nAfter concatenations:\n";
22 string s4{s1 + "apult"}; // concatenation
23 s1.append("acomb"); // create "catacomb"
24 s3 += "pet"; // create "carpet" with overloaded +=
25 cout << fmt::format("s1: {}\ns3: {}\ns4: {}\n", s1, s3, s4);
26
27 // append locations 4 through end of s1 to
28 // create string "comb" (s5 was initially empty)
29 string s5; // initialized to the empty string
30 s5.append(s1, 4, s1.size() - 4);
31 cout << fmt::format("s5: {}", s5);
32 }
```

After concatenations:

s1: catacomb
s3: carpet
s4: catapult
s5: comb

```
28 // comparing s1 and s4
29 if (s1 > s4) {
30     cout << "\n\ns1 > s4\n";
31 }
32
```

s1 > s4

```
33 // comparing s1 and s2
34 displayResult("s1.compare(s2)", s1.compare(s2));
35
36 // comparing s1 (elements 2-5) and s3 (elements 0-5)
37 displayResult("s1.compare(2, 5, s3, 0, 5)",
38     s1.compare(2, 5, s3, 0, 5));
39
40 // comparing s2 and s4
41 displayResult("s4.compare(0, s2.size(), s2)",
42     s4.compare(0, s2.size(), s2));
43
44 // comparing s2 and s4
45 displayResult("s2.compare(0, 3, s4)", s2.compare(0, 3, s4));
46 }
```

```
s1.compare(s2) > 0
s1.compare(2, 5, s3, 0, 5) == 0
s4.compare(0, s2.size(), s2) == 0
s2.compare(0, 3, s4) < 0
```

```
15     string string1; // empty string
16
17     cout << "Statistics before input:\n";
18     printStatistics(string1);
19
```

```
Statistics before input:
capacity: 15
max size: 2147483647
size: 0
empty: true
```

```
20 cout << "\n\nEnter a string: ";
21 cin >> string1; // delimited by whitespace
22 cout << "The string entered was: " << string1;
23 cout << "\nStatistics after input:\n";
24 printStatistics(string1);
25
```

```
Enter a string: tomato
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 2147483647
size: 6
empty: false
```

```
26     cout << "\n\nEnter a string: ";
27     cin >> string1; // delimited by whitespace
28     cout << "The string entered was: " << string1;
29     cout << "\nStatistics after input:\n";
30     printStatistics(string1);
31
```

```
Enter a string: soup
The string entered was: soup
Statistics after input:
capacity: 15
max size: 2147483647
size: 4
empty: false
```

```
32 // append 46 characters to string1
33 string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
34 cout << "\n\nstring1 is now: " << string1;
35 cout << "\nStatistics after concatenation:\n";
36 printStatistics(string1);
37
```

```
Statistics after resizing to add 10 characters:
capacity: 63
max size: 2147483647
size: 60
empty: false
```

```
38     string1.resize(string1.size() + 10); // add 10 elements to string1
39     cout << "\n\nStatistics after resizing to add 10 characters:\n";
40     printStatistics(string1);
41     cout << endl;
42 }
```

```
Statistics after resizing to add 10 characters:
capacity: 63
max size: 2147483647
size: 60
empty: false
```

```
12 // find "is" from the beginning and end of s
13 cout << fmt::format("\ns.find(\"is\"): {}\n.s.rfind(\"is\"): {}",
14                 s.find("is"), s.rfind("is")));
15
```

```
s.find("is"): 5
s.rfind("is"): 23
```

```
16 // find 'o' from beginning
17 int location = s.find_first_of("misop");
18 cout << fmt::format("\ns.find_first_of(\"misop\") found {} at {}", 
19                   s.at(location), location);
20
```

s.find_first_of("misop") found o at 1

```
21 // find 'o' from end
22 location = s.find_last_of("misop");
23 cout << fmt::format("\ns.find_last_of(\"misop\") found {} at {}", 
24                   s.at(location), location);
25
```

```
s.find_last_of("misop") found o at 27
```

```
26 // find '1' from beginning
27 location = s.find_first_not_of("noi spm");
28 cout << fmt::format(
29     "\ns.find_first_not_of(\"noi spm\") found {} at {}",
30     s.at(location), location);
31
32 // find '.' at location 13
33 location = s.find_first_not_of("12noi spm");
34 cout << fmt::format(
35     "\ns.find_first_not_of(\"12noi spm\") found {} at {}",
36     s.at(location), location);
37
38 // search for characters not in "noon is 12pm; midnight is not"
39 location = s.find_first_not_of("noon is 12pm; midnight is not");
40 cout << fmt::format("\ns.find_first_not_of(" +
41     "\"noon is 12pm; midnight is not\"): {}", location);
42 }
```

```
s.find_first_not_of("noi spm") found 1 at 8
s.find_first_not_of("12noi spm") found ; at 12
s.find_first_not_of("noon is 12pm; midnight is not"): -1
```

```
18     string1.erase(62); // remove from index 62 through end of string1
19     cout << fmt::format("string1 after erase:\n{}\n\n", string1);
20
```

string1 after erase:
The values in any left subtree
are less than the value in the

```
21 size_t position = string1.find(" "); // find first space
22
23 // replace all spaces with period
24 while (position != string::npos) {
25     string1.replace(position, 1, ".");
26     position = string1.find(" ", position + 1);
27 }
28
29 cout << fmt::format("After first replacement:\n{}\n\n", string1);
30
```

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

```
31 position = string1.find("."); // find first period
32
33 // replace all periods with two semicolons
34 // NOTE: this will overwrite characters
35 while (position != string::npos) {
36     string1.replace(position, 2, "xxxxx; ;yyy", 5, 2);
37     position = string1.find(".", position + 1);
38 }
39
40 cout << fmt::format("After second replacement:\n{}\n", string1);
41 }
```

After first replacement:

The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:

The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he

```
int convertedInt = stoi(s, nullptr, 10);
```

```
int convertedInt = stoi(s, &index, 2);
```

```
15 // string_views see changes to the characters they view
16 s1.at(0) = 'R'; // capitalize s1
17 cout << fmt::format("s1: {}\ns2: {}\n{}1: {}\\n\\n", s1, s2, v1);
18
```

```
s1: Red
s2: red
v1: Red
```

```
19 // string_views are comparable with strings or string_views
20 cout << fmt::format("s1 == v1: {}\n s2 == v1: {}\n\n",
21                     s1 == v1, s2 == v1);
22
```

```
s1 == v1: true
s2 == v1: false
```

```
23 // string_view can remove a prefix or suffix
24 v1.remove_prefix(1); // remove one character from the front
25 v1.remove_suffix(1); // remove one character from the back
26 cout << fmt::format("s1: {}\n v1: {}\n\n", s1, v1);
27
```

s1: Red
v1: e

```
28 // string_views are iterable
29 string_view v2{"C-string"};
30 cout << "The characters in v2 are: ";
31 for (const char c : v2) {
32     cout << c << " ";
33 }
34
```

The characters in v2 are: C - s t r i n g

```
35 // string_views enable various string operations on C-Strings
36 cout << fmt::format("\n\nv2.size(): {}", v2.size());
37 cout << fmt::format("v2.find('-'): {}", v2.find('-'));
38 cout << fmt::format("v2.starts_with('C'): {}", v2.starts_with('C'));
39 }
```

```
v2.size(): 8
v2.find('-'): 1
v2.starts_with('C'): true
```

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);

// position n bytes in fileObject
fileObject.seekg(n, ios::cur);

// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);

// position at end of fileObject
fileObject.seekg(0, ios::end);
```

```
input >> account >> quoted(name) >> balance
```

```
string windowsPath{"C:\\MyFolder\\MySubFolder\\MyFile.txt"}
```

```
string windowsPath{R"(C:\MyFolder\MySubFolder\MyFile.txt)"}  
;
```

R"MYDELIMITER(.)*\d[0-35-9]-\d\d-\d\d)MYDELIMITER"

```
20 // get data by column; ignoring column 0 in this example
21 auto survived{titanic.GetColumn<int>(R("survived"))};
22 auto sex{titanic.GetColumn<string>(R("sex"))};
23 auto age{titanic.GetColumn<double>(R("age"))};
24 auto pclass{titanic.GetColumn<int>(R("pclass"))};
25
```

```
26 // lambda to remove the quotes from the strings
27 for (string& item : sex) {
28     item.erase(0, 1).pop_back();
29 }
30
```

```
31 // display first 5 rows
32 cout << fmt::format("First five rows:\n{:<10}{:<8}{:<6}{}\n",
33                 "survived", "sex", "age", "class");
34 for (size_t i{0}; i < 5; ++i) {
35     cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
36             survived.at(i), sex.at(i), age.at(i), passengerClass.at(i));
37 }
38
```

```
First five rows:
survived  sex      age   class
1         female   29.0  1
1         male     0.9   1
0         female   2.0   1
0         male     30.0  1
0         female   25.0  1
```

```
39 // display last 5 rows
40 cout << fmt::format("\nLast five rows:\n{:<10}{:<8}{:<6}{}\n",
41     "survived", "sex", "age", "class");
42 auto count{titanic.GetRowCount()};
43 for (size_t i{count - 5}; i < count; ++i) {
44     cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
45         survived.at(i), sex.at(i), age.at(i), pclass.at(i));
46 }
47
```

Last five rows:

	survived	sex	age	class
0		female	14.5	3
0		female	nan	3
0		male	26.5	3
0		male	27.0	3
0		male	29.0	3

```
48 // use C++20 ranges to eliminate missing values from age column
49 auto removeNaN =
50     age | views::filter([](const auto& x) {return !isnan(x);});
51 vector<double> cleanAge{begin(removeNaN), end(removeNaN)};
52
```

```
53 // descriptive statistics for cleaned ages column
54 sort(begin(cleanAge), end(cleanAge));
55 size_t size{cleanAge.size()};
56 double median{};
57
58 if (size % 2 == 0) { // find median value for even number of items
59     median = (cleanAge.at(size / 2 - 1) + cleanAge.at(size / 2)) / 2;
60 }
61 else { // find median value for odd number of items
62     median = cleanAge.at(size / 2);
63 }
64
65 cout << "\nDescriptive statistics for the age column:\n"
66     << fmt::format("Passengers with age data: {}\n", size)
67     << fmt::format("Average age: {:.2f}\n",
68                     accumulate(begin(cleanAge), end(cleanAge), 0.0) / size)
69     << fmt::format("Minimum age: {:.2f}\n", cleanAge.front())
70     << fmt::format("Maximum age: {:.2f}\n", cleanAge.back())
71     << fmt::format("Median age: {:.2f}\n", median);
72
```

Descriptive statistics for the age column:

Passengers with age data: 1046

Average age: 29.88

Minimum age: 0.17

Maximum age: 80.00

Median age: 28.00

```
73 // passenger counts by class
74 auto countClass = [](const auto& column, const int classNumber) {
75     auto filterByClass{column
76         | views::filter([classNumber](auto x) {return classNumber == x;})
77         | views::transform([](auto x) {return 1;})};
78     return accumulate(begin(filterByClass), end(filterByClass), 0);
79 };
80
81 constexpr int firstClass{1};
82 constexpr int secondClass{2};
83 constexpr int thirdClass{3};
84 const int firstCount{countClass(pclass, firstClass)};
85 const int secondCount{countClass(pclass, secondClass)};
86 const int thirdCount{countClass(pclass, thirdClass)};
87
```

```
88     cout << "\nPassenger counts by class:\n"
89     << fmt::format("1st: {}\\n2nd: {}\\n3rd: {}\\n\\n",
90                  firstCount, secondCount, thirdCount);
91
```

Passenger counts by class:

1st: 323
2nd: 554
3rd: 2127

```
92 // percentage of people who survived
93 auto survivors = survived | views::filter([](auto x) {return x;});
94 int survivorCount{accumulate(begin(survivors), end(survivors), 0)};
95
96 cout << fmt::format("\nSurvived count: {}\nDied count: {}",
97                     survivorCount, survived.size() - survivorCount);
98 cout << fmt::format("Percent who survived: {:.2f}%\n\n",
99                     100.0 * survivorCount / survived.size());
100
```

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
```

```
101 // count who survived by male/female, 1st/2nd/3rd class
102 int survivingMen{0};
103 int survivingWomen{0};
104 int surviving1st{0};
105 int surviving2nd{0};
106 int surviving3rd{0};
```

```
I07
I08     for (size_t i{0}; i < survived.size(); ++i) {
I09         if (survived.at(i)) {
I10             sex.at(i) == "female"s ? ++survivingWomen : ++survivingMen;
I11
I12             if (firstClass == pclass.at(i)) {
I13                 ++surviving1st;
I14             }
I15             else if (secondClass == pclass.at(i)) {
I16                 ++surviving2nd;
I17             }
I18             else { // third class
I19                 ++surviving3rd;
I20             }
I21         }
I22     }
I23
```

```
I24 // percentages who survived by male/female, 1st/2nd/3rd class
I25 cout << fmt::format("Female survivor percentage: {:.2f}%\n",
I26             100.0 * survivingWomen / survivorCount)
I27             << fmt::format("Male survivor percentage: {:.2f}%\n\n",
I28             100.0 * survivingMen / survivorCount)
I29             << fmt::format("1st class survivor percentage: {:.2f}%\n",
I30             100.0 * surviving1st / survivorCount)
I31             << fmt::format("2nd class survivor percentage: {:.2f}%\n",
I32             100.0 * surviving2nd / survivorCount)
I33             << fmt::format("3rd class survivor percentage: {:.2f}%\n",
I34             100.0 * surviving3rd / survivorCount);
I35 }
```

Female survivor percentage: 67.80%

Male survivor percentage: 32.20%

1st class survivor percentage: 40.00%

2nd class survivor percentage: 23.80%

3rd class survivor percentage: 36.20%

```
15 // fully match five digits
16 regex r2(R"(\d{5})");
17 cout << R"(Matching against: \d{5})" << "\n"
18     << fmt::format("02215: {}; 9876: {}\\n\\n",
19                     regex_match("02215", r2), regex_match("9876", r2));
20
```

```
Matching against: \d{5}
02215: true; 9876: false
```

```
21 // match a word that starts with a capital letter
22 regex r3("[A-Z][a-z]*");
23 cout << "Matching against: [A-Z][a-z]*\n"
24     << fmt::format("Wally: {}; eva: {}\\n\\n",
25                 regex_match("Wally", r3), regex_match("eva", r3));
26
```

```
Matching against: [A-Z][a-z]*
Wally: true; eva: false
```

```
27 // match any character that's not a lowercase letter
28 regex r4("[^a-z]");
29 cout << "Matching against: [^a-z]\n"
30     << fmt::format("A: {}; a: {}\\n\\n",
31                     regex_match("A", r4), regex_match("a", r4));
32
```

Matching against: [^a-z]

A: true; a: false

```
33 // match metacharacters as literals in a custom character class
34 regex r5("[*+$]");
35 cout << "Matching against: [*+$]\n"
36     << fmt::format("*: {}; !: {}\n\n",
37                     regex_match("*", r5), regex_match("!", r5));
38
```

```
Matching against: [*+$]
*: true; !: false
```

```
39 // matching a capital letter followed by at least one lowercase letter
40 regex r6("[A-Z][a-z]+");
41 cout << "Matching against: [A-Z][a-z]*\n"
42     << fmt::format("Wally: {}; E: {}\\n\\n",
43                     regex_match("Wally", r6), regex_match("E", r6));
44
```

```
Matching against: [A-Z][a-z]*
Wally: true; E: false
```

```
45 // matching zero or one occurrences of a subexpression
46 regex r7("label1?ed");
47 cout << "Matching against: label1?ed\n"
48     << fmt::format("labelled: {}; labeled: {}; labelled: {}\n\n",
49                 regex_match("labelled", r7), regex_match("labeled", r7),
50                 regex_match("label1led", r7));
51
```

Matching against: labelled?
labelled: true; labeled: true; labelled: false

```
52 // matching n (3) or more occurrences of a subexpression
53 regex r8(R"(\d{3,})");
54 cout << R"(Matching against: \d{3,})" << "\n"
55     << fmt::format("123: {}; 1234567890: {}; 12: {}\\n\\n",
56                     regex_match("123", r8), regex_match("1234567890", r8),
57                     regex_match("12", r8));
58
```

```
Matching against: \d{3,}
123: true; 1234567890: true; 12: false
```

```
59 // matching n to m inclusive (3-6), occurrences of a subexpression
60 regex r9(R"(\d{3,6})");
61 cout << R"(Matching against: \d{3,6})" << "\n"
62     << fmt::format("123: {}; 123456: {}; 1234567: {}; 12: {}\\n",
63                 regex_match("123", r9), regex_match("123456", r9),
64                 regex_match("1234567", r9), regex_smatch("12", r9));
65 }
```

```
Matching against: \d{3,6}\
123: true; 123456: true; 1234567: false; 12: false
```

```
19 // ignoring case
20 string s2{"SAM WHITE"};
21 smatch match; // store the text that matches the pattern
22 cout << fmt::format("s2: {}\n\n", s2);
23 cout << "Case insensitive search for Sam in s2:\n"
24     << fmt::format("Sam: {}\n",
25                 regex_search(s2, match, regex{"Sam", regex_constants::icase}));
26     << fmt::format("Matched text: {}\n\n", match.str());
27
```

```
s2: SAM WHITE
```

```
Case insensitive search for Sam in s2:
```

```
Sam: true
```

```
Matched text: SAM
```

```
28 // finding all matches
29 string contact{"Wally White, Home: 555-555-1234, Work: 555-555-4321"};
30 regex phone{R"(\d{3}-\d{3}-\d{4})"};
31
32 cout << "\nFinding all phone numbers in:\n" << contact << "\n";
33 while (regex_search(contact, match, phone)) {
34     cout << fmt::format("  {}\\n", match.str());
35     contact = match.suffix();
36 }
37 }
```

Finding all phone numbers in:
Wally White, Home: 555-555-1234, Work: 555-555-4321
555-555-1234
555-555-4321

```
1 // fig08_01.cpp
2 // Demonstrating string assignment and concatenation.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
```

```
1 // fig08_02.cpp
2 // Comparing strings.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 void displayResult(const string& s, int result) {
9     if (result == 0) {
10         cout << s + " == 0\n";
11     }
12     else if (result > 0) {
13         cout << s + " > 0\n";
14     }
15     else { // result < 0
16         cout << s + " < 0\n";
17     }
18 }
19
20 int main() {
21     const string s1{"Testing the comparison functions."};
22     const string s2{"Hello"};
23     const string s3{"stinger"};
24     const string s4{s2}; // "Hello"
25
26     cout << fmt::format("s1: {}\ns2: {}\ns3: {}\ns4: {}", s1, s2, s3, s4);
27 }
```

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
s4: Hello
```

```
1 // fig08_03.cpp
2 // Demonstrating string member function substr.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main() {
8     const string s{"airplane"};
9     cout << s.substr(3, 4) << endl; // retrieve substring "plan"
10 }
```

```
plan
```

```
1 // fig08_04.cpp
2 // Using the swap function to swap two strings.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
9     string first{"one"};
10    string second{"two"};
11
12    cout << fmt::format(
13        "Before swap:\nfirst: {}; second: {}", first, second);
14    first.swap(second); // swap strings
15    cout << fmt::format(
16        "\n\nAfter swap:\nfirst: {}; second: {}", first, second);
17 }
```

```
Before swap:
first: one; second: two
```

```
After swap:
first: two; second: one
```

```
1 // fig08_05.cpp
2 // Printing string characteristics.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 // display string statistics
9 void printStatistics(const string& s) {
10     cout << fmt::format("capacity: {}\nmax size: {}\nsize: {}\nempty: {}",
11                         s.capacity(), s.max_size(), s.size(), s.empty());
12 }
13
14 int main() {
```

```
1 // fig08_06.cpp
2 // Demonstrating the string find member functions.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
9     const string s{"noon is 12pm; midnight is not"};
10    cout << "Original string: " << s;
11 }
```

Original string: noon is 12pm; midnight is not

```
1 // fig08_07.cpp
2 // Demonstrating string member functions erase and replace.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
9     // compiler concatenates all parts into one string
10    string string1{"The values in any left subtree"
11        "\nare less than the value in the"
12        "\nparent node and the values in"
13        "\nany right subtree are greater"
14        "\nthan the value in the parent node"};
15
16    cout << fmt::format("Original string:\n{}\\n\\n", string1);
17
```

Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

```
1 // fig08_08.cpp
2 // Demonstrating std::string insert member functions.
3 #include <iostream>
4 #include <string>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
9     string s1{"beginning end"};
10    string s2{"12345678"};
11
12    cout << fmt::format("Initial strings:\n{s1}\n{s2}\n", s1, s2);
13
14    s1.insert(10, "middle "); // insert "middle " at location 10 in s1
15    s2.insert(3, "xx", 0, string::npos); // insert "xx" at location 3 in s2
16
17    cout << fmt::format("Strings after insert:\n{s1}\n{s2}\n", s1, s2);
18 }
```

Initial strings:
s1: beginning end
s2: 12345678

Strings after insert:
s1: beginning middle end
s2: 123xx45678

```
1 // fig08_09.cpp
2 // C++17 string_view.
3 #include <iostream>
4 #include <string>
5 #include <string_view>
6 #include "fmt/format.h" // In C++20, this will be #include <format>
7 using namespace std;
8
9 int main() {
10     string s1{"red"};
11     string s2{s1};
12     string_view v1{s1}; // v2 "sees" the contents of s1
13     cout << fmt::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
14 }
```

```
s1: red
s2: red
v1: red
```

```
1 // fig08_10.cpp
2 // Creating a sequential file.
3 #include <cstdlib> // exit function prototype
4 #include <fstream> // contains file stream processing types
5 #include <iostream>
6 #include <string>
7 #include "fmt/format.h" // In C++20, this will be #include <format>
8 using namespace std;
9
10 int main() {
11     // ofstream opens the file
12     if (ofstream output{"clients.txt", ios::out}; output) {
13         cout << "Enter the account, name, and balance.\n"
14             << "Enter end-of-file to end input.\n" ;
```

```
15     int account;
16     string name;
17     double balance;
18
19     // read account, name and balance from cin, then place in file
20     while (cin >> account >> name >> balance) {
21         output << fmt::format("{} {} {}\n", account, name, balance);
22     }
23 }
24 else {
25     cerr << "File could not be opened\n";
26     exit(EXIT_FAILURE);
27 }
28 }
29 }
```

Enter the account, name, and balance.

Enter end-of-file to end input.

```
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

```
1 // fig08_11.cpp
2 // Reading and printing a sequential file.
3 #include <cstdlib>
4 #include <fstream> // file stream
5 #include <iostream>
6 #include <string>
7 #include "fmt/format.h" // In C++20, this will be #include <format>
8 using namespace std;
9
```

```

10 int main() {
11     // ifstream opens the file
12     if (ifstream input{"clients.txt", ios::in}; input) {
13         cout << fmt::format("{:<10}{:<13}{}\n",
14                             "Account", "Name", "Balance");
15
16         int account;
17         string name;
18         double balance;
19
20         // display each record in file
21         while (input >> account >> name >> balance) {
22             cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
23                                 account, name, balance);
24         }
25     }
26     else {
27         cerr << "File could not be opened\n";
28         exit(EXIT_FAILURE);
29     }
30 }
```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

```

1 // fig08_12.cpp
2 // Using an ostringstream object.
3 #include <iostream>
4 #include <sstream> // header for string stream processing
5 #include <string>
6 using namespace std;
7
8 int main() {
9     ostringstream output; // create ostringstream object
10 }
```

```
11 const string string1{"Output of several data types "};
12 const string string2{"to an ostringstream object:"};
13 const string string3{"\n      double: "};
14 const string string4{"\n      int: "};
15
16 constexpr double s{123.4567};
17 constexpr int i{22};
18
19 // output strings, double and int to ostringstream
20 output << string1 << string2 << string3 << d << string4 << i;
21
22 // call str to obtain string contents of the ostringstream
23 cout << "output contains:\n" << output.str();
24
25 // add additional characters and call str to output string
26 output << "\nmore characters added";
27 cout << "\n\nafter additional stream insertions, output contains:\n"
28     << "" << output.str() << endl;
29 }
```

```
outputString contains:
Output of several data types to an ostringstream object:
double: 123.457
int: 22
```

```
after additional stream insertions, outputString contains:
Output of several data types to an ostringstream object:
double: 123.457
int: 22
more characters added
```

```
1 // fig08_13.cpp
2 // Demonstrating input from an istringstream object.
3 #include <iostream>
4 #include <sstream>
5 #include <string>
6 #include "fmt/format.h" // In C++20, this will be #include <format>
7 using namespace std;
8
9 int main() {
10    const string inputString{"Amanda test 123 4.7 A"};
11    istringstream input{inputString};
12    string s1;
13    string s2;
14    int i;
15    double d;
16    char c;
17
18    input >> s1 >> s2 >> i >> d >> c;
19
20    cout << "Items extracted from the istringstream object:\n"
21        << fmt::format("{}\n{}\n{}\n{}\n{}\n{}", s1, s2, i, d, c);
22
23    // attempt to read from empty stream
24    if (long value; input >> value) {
25        cout << fmt::format("\n\nlong value is: {}\n", value);
26    }
27    else {
28        cout << fmt::format("\n\ninput is empty\n");
29    }
30 }
```

Items extracted from the istringstream object:

Amanda

test

123

4.7

A

input is empty

```
1 // fig08_14.cpp
2 // Reading from a CSV file.
3 #include <iostream>
4 #include <string>
5 #include <vector>
6 #include "fmt/format.h" // In C++20, this will be #include <format>
7 #include "rapidcsv.h"
8 using namespace std;
9
10 int main() {
11     rapidcsv::Document document{"accounts.csv"}; // loads accounts.csv
12     vector<int> accounts{document.GetColumn<int>("account")};
13     vector<string> names{document.GetColumn<string>("name")};
14     vector<double> balances{document.GetColumn<double>("balance")};
15
16     cout << fmt::format("{:<10}{:<13}{}\n", "Account", "Name", "Balance");
17
18     for (size_t i{0}; i < accounts.size(); ++i) {
19         cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
20                             accounts.at(i), names.at(i), balances.at(i));
21     }
22 }
```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

```
1 // fig08_14.cpp
2 // Reading the Titanic dataset from a CSV file, then analyzing it.
3 #include <algorithm>
4 #include <cmath>
5 #include <iostream>
6 #include <numeric>
7 #include <ranges>
8 #include <string>
9 #include <vector>
10 #include "fmt/format.h" // In C++20, this will be #include <format>
11 #include "rapidcsv.h"
12 using namespace std;
13
14 int main() {
15     // Load Titanic dataset; treat missing age values as NaN
16     rapidcsv::Document titanic("titanic.csv",
17         rapidcsv::LabelParams{}, rapidcsv::SeparatorParams{},
18         rapidcsv::ConverterParams{true});
19 }
```

```
1 // fig08_16.cpp
2 // Matching entire strings to regular expressions.
3 #include <iostream>
4 #include <regex>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 using namespace std;
7
8 int main() {
9     // fully match a pattern of literal characters
10    regex r1("02215");
11    cout << "Matching against: 02215\n"
12    << fmt::format("02215: {}; 51220: {}\n\n",
13        regex_match("02215", r1), regex_match("51220", r1));
14 }
```

Matching against: 02215
02215: true; 51220: false

```
1 // fig08_17.cpp
2 // Regular expression replacements and splitting.
3 #include <iostream>
4 #include <regex>
5 #include <string>
6 #include <vector>
7 #include "fmt/format.h" // In C++20, this will be #include <format>
8 using namespace std;
9
10 int main() {
11     // replace tabs with commas
12     string s1("1\t2\t3\t4");
13     cout << "Original string: 1\t2\t3\t4\n";
14     cout << fmt::format("After replacing tabs with commas: {}\\n",
15                         regex_replace(s1, regex{R"(\t)"}, ","));
16 }
```

```
Original string: 1\t2\t3\t4
After replacing tabs with commas: 1,2,3,4
```

```
1 // fig08_18.cpp
2 // Matching patterns throughout a string.
3 #include <iostream>
4 #include <regex>
5 #include <string>
6 #include "fmt/format.h" // In C++20, this will be #include <format>
7 using namespace std;
8
```

```
9 int main() {
10 // performing a simple match
11 string s1{"Programming is fun"};
12 cout << fmt::format("s1: {}\n\n", s1);
13 cout << "Search anywhere in s1:\n"
14     << fmt::format("Programming: {}; fun: {}; fn: {}\\n\\n",
15                 regex_search(s1, regex{"Programming"}),
16                 regex_search(s1, regex{"fun"}),
17                 regex_search(s1, regex{"fn"}));
18 }
```

```
s1: Programming is fun
```

```
Search anywhere in s1:
```

```
Programming: true; fun: true; fn: false
```

```
1 // Fig. 9.1: AccountTest.cpp
2 // Creating and manipulating an Account object.
3 #include <iostream>
4 #include <string>
5 #include <fmt/format.h> // In C++20, this will be #include <format>
6 #include "Account.h"
7
8 using namespace std;
9
10 int main() {
11     Account myAccount{}; // create Account object myAccount
12
13     // show that the initial value of myAccount's name is the empty string
14     cout << fmt::format("Initial account name: {}\n", myAccount.getName());
15
16     // prompt for and read the name
17     cout << "Enter the account name: ";
18     string name{};
19     getline(cin, name); // read a line of text
20     myAccount.setName(name); // put name in the myAccount object
21
22     // display the name stored in object myAccount
23     cout << fmt::format("Updated account name: {}\n", myAccount.getName());
24 }
```

```
Initial account name:
Enter the account name: Jane Green
Updated account name: Jane Green
```

```
1 // Fig. 9.2: Account.h
2 // Account class with a data member and
3 // member functions to set and get its value.
4 #include <string>
5 #include <string_view>
6
7 class Account {
8 public:
9     // member function that sets m_name in the object
10    void setName(std::string_view name) {
11        m_name = name; // replace m_name's value with name
12    }
13
14    // member function that retrieves the account name from the object
15    const std::string& getName() const {
16        return m_name; // return m_name's value to this function's caller
17    }
18 private:
19     std::string m_name; // data member containing account holder's name
20 }; // end class Account
```

```
Account myAccount{}; // create Account object myAccount
std::string m_name; // data member containing account holder's name
void setName(std::string_view name) {
    m_name = name; // replace m_name's value with name
}
const std::string& getName() const {
    return m_name; // return m_name's value to this function's caller
}
Account account1{"Jane Green"};
```

```
1 // Fig. 9.3: Account.h
2 // Account class with a constructor that initializes the account name.
3 #include <string>
4 #include <string_view>
5
6 class Account {
7 public:
8     // constructor initializes data member m_name with the parameter name
9     explicit Account(std::string_view name)
10        : m_name{name} { // member initializer
11            // empty body
12    }
13
14    // function to set the account name
15    void setName(std::string_view name) {
16        m_name = name; // replace m_name's value with name
17    }
18
19    // function to retrieve the account name
20    const std::string& getName() const {
21        return m_name;
22    }
23 private:
24     std::string m_name; // account name data member
25 }; // end class Account
```

```
explicit Account(std::string_view name)
: m_name{name} { // member initializer
// empty body
}
Account account1{"Jane Green"};
Account account1{"Jane Green"};
```

```
1 // Fig. 9.4: AccountTest.cpp
2 // Using the Account constructor to initialize the m_name data
3 // member at the time each Account object is created.
4 #include <iostream>
5 #include <fmt/format.h> // In C++20, this will be #include <format>
6 #include "Account.h"
7
8 using namespace std;
9
10 int main() {
11     // create two Account objects
12     Account account1{"Jane Green"};
13     Account account2{"John Blue"};
14
15     // display initial each Account's corresponding name
16     cout << fmt::format("account1 name is: {}\\naccount2 name is: {}\\n",
17                         account1.getName(), account2.getName());
18 }
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

```
Account account2{"John Blue"};
```

```
1 // Fig. 9.5: Account.h
2 // Account class with m_name and m_balance data members, and a
3 // constructor and deposit function that each perform validation.
4 #include <string>
5 #include <string_view>
6
7 class Account {
8 public:
9     // Account constructor with two parameters
10    Account(std::string_view name, double balance)
11        : m_name{name} { // member initializer for m_name
12
13        // validate that balance is greater than 0.0; if not,
14        // data member m_balance keeps its default initial value of 0.0
15        if (balance > 0.0) { // if the balance is valid
16            m_balance = balance; // assign it to data member m_balance
17        }
18    }
```

```
19 // function that deposits (adds) only a valid amount to the balance
20 void deposit(double amount) {
21     if (amount > 0.0) { // if the amount is valid
22         m_balance += amount; // add it to m_balance
23     }
24 }
25 }

26 // function returns the account balance
27 double getBalance() const {
28     return m_balance;
29 }
30 }

31 // function that sets the account name
32 void setName(std::string_view name) {
33     m_name = name; // replace m_name's value with name
34 }
35 }

36 // function that returns the account name
37 const std::string& getName() const {
38     return m_name;
39 }
40 }

41 private:
42     std::string m_name; // account name data member
43     double m_balance{0.0}; // data member with default initial value
44 }; // end class Account
```

```
double m_balance{0.0}; // data member with default initial value
if (balance > 0.0) { // if the balance is valid
    m_balance = balance; // assign it to data member m_balance
}
```

```
1 // Fig. 9.6: AccountTest.cpp
2 // Displaying and updating Account balances.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Account.h"
6
7 using namespace std;
8
9 int main() {
10    Account account1{"Jane Green", 50.00};
11    Account account2{"John Blue", -7.00};
12
13    // display initial balance of each object
14    cout << fmt::format("account1: {} balance is ${:.2f}\n",
15                        account1.getName(), account1.getBalance());
16    cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
17                        account2.getName(), account2.getBalance());
18}
```

```
account1: Jane Green balance is $50.00
account2: John Blue balance is $0.00
```

```
19    cout << "Enter deposit amount for account1: " // prompt
20    double amount;
21    cin >> amount; // obtain user input
22    cout << fmt::format("adding ${:.2f} to account1 balance\n\n", amount);
23    account1.deposit(amount); // add to account1's balance
24
25    // display balances
26    cout << fmt::format("account1: {} balance is ${:.2f}\n",
27                        account1.getName(), account1.getBalance());
28    cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
29                        account2.getName(), account2.getBalance());
30
```

```
Enter deposit amount for account1: 25.37
adding $25.37 to account1 balance

account1: Jane Green balance is $75.37
account2: John Blue balance is $0.00
```

```
31     cout << "Enter deposit amount for account2: " // prompt
32     cin >> amount; // obtain user input
33     cout << fmt::format("adding ${:.2f} to account2 balance\n\n", amount);
34     account2.deposit(amount); // add to account2 balance
35
36     // display balances
37     cout << fmt::format("account1: {} balance is ${:.2f}\n",
38                         account1.getName(), account1.getBalance());
39     cout << fmt::format("account2: {} balance is ${:.2f}\n",
40                         account2.getName(), account2.getBalance());
41 }
```

```
Enter deposit amount for account2: 123.45
adding $123.45 to account2 balance
```

```
account1: Jane Green balance is $75.37
account2: John Blue balance is $123.45
```

```
1 // Fig. 9.7: Time.h
2 // Time class definition.
3 // Member functions are defined in Time.cpp
4 #pragma once // prevent multiple inclusions of header
5 #include <string>
6
7 // Time class definition
8 class Time {
9 public:
10    void setTime(int hour, int minute, int second);
11    std::string to24HourString() const; // 24-hour string format
12    std::string to12HourString() const; // 12-hour string format
13 private:
14    int m_hour{0}; // 0 - 23 (24-hour clock format)
15    int m_minute{0}; // 0 - 59
16    int m_second{0}; // 0 - 59
17};
```

```
1 // Fig. 9.8: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept> // for invalid_argument exception class
4 #include <string>
5 #include <fmt/format.h> // In C++20, this will be #include <format>
6 #include "Time.h" // include definition of class Time from Time.h
7
8 using namespace std;
9
10 // set new Time value using 24-hour time
11 void Time::setTime(int hour, int minute, int second) {
12     // validate hour, minute and second
13     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
14         (second < 0 || second >= 60)) {
15         throw invalid_argument{"hour, minute or second was out of range"};
16     }
17
18     m_hour = hour;
19     m_minute = minute;
20     m_second = second;
21 }
22
23 // return Time as a string in 24-hour format (HH:MM:SS)
24 string Time::to24HourString() const {
25     return fmt::format("{:02d}:{:02d}:{:02d}", m_hour, m_minute, m_second);
26 }
27
28 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
29 string Time::to12HourString() const {
30     return fmt::format("{}:{}{:02d} {}",
31                         ((m_hour % 12 == 0) ? 12 : m_hour % 12), m_minute, m_second,
32                         (m_hour < 12 ? "AM" : "PM"));
33 }
```

```
1 // fig09_09.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with Time.cpp.
4 #include <iostream>
5 #include <stdexcept> // invalid_argument exception class
6 #include <string_view>
7 #include <fmt/format.h> // In C++20, this will be #include <format>
8 #include "Time.h" // definition of class Time from Time.h
9 using namespace std;
10
11 // displays a Time in 24-hour and 12-hour formats
12 void displayTime(string_view message, const Time& time) {
13     cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}"
14                         "\n", message, time.to24HourString(), time.to12HourString());
15 }
16
17 int main() {
18     Time t{}; // instantiate object t of class Time
19
20     displayTime("Initial time:", t); // display t's initial value
21     t.setTime(13, 27, 6); // change time
22     displayTime("After setTime:", t); // display t's new value
```

```

23
24 // attempt to set the time with invalid values
25 try {
26     t.setTime(99, 99, 99); // all values out of range
27 }
28 catch (const invalid_argument& e) {
29     cout << fmt::format("Exception: {}\n\n", e.what());
30 }
31
32 // display t's value after attempting to set an invalid time
33 displayTime("After attempting to set an invalid time:", t);
34 }
```

Initial time:
 24-hour time: 00:00:00
 12-hour time: 12:00:00 AM

After setTime:
 24-hour time: 13:27:06
 12-hour time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
 24-hour time: 13:27:06
 12-hour time: 1:27:06 PM

```

Time sunset{}; // object of type Time
array<Time, 5> arrayOfTimes{}; // std::array of 5 Time objects
Time& dinnerTimeRef{sunset}; // reference to a Time object
Time* timePtr{&sunset}; // pointer to a Time object
Account account{}; // an Account object
Account& ref{account}; // ref refers to an Account object
Account* ptr{&account}; // ptr points to an Account object
account.deposit(123.45); // call deposit via account object
ref.deposit(123.45); // call deposit via reference to account
ptr->deposit(123.45); // call deposit via pointer to account
```

```
1 // Fig. 9.10: Time.h
2 // Time class containing a constructor with default arguments.
3 // Member functions defined in Time.cpp.
4 #pragma once // prevent multiple inclusions of header
5 #include <string>
6
7 // Time class definition
8 class Time {
9 public:
10    // default constructor because it can be called with no arguments
11    explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13    // set functions
14    void setTime(int hour, int minute, int second);
15    void setHour(int hour); // set hour (after validation)
16    void setMinute(int minute); // set minute (after validation)
17    void setSecond(int second); // set second (after validation)
18
19    // get functions
20    int getHour() const; // return hour
21    int getMinute() const; // return minute
22    int getSecond() const; // return second
23
24    std::string to24HourString() const; // 24-hour time format string
25    std::string to12HourString() const; // 12-hour time format string
26 private:
27    int m_hour{0}; // 0 - 23 (24-hour clock format)
28    int m_minute{0}; // 0 - 59
29    int m_second{0}; // 0 - 59
30};
```

```
1 // Fig. 9.11: Time.cpp
2 // Member-function definitions for class Time.
3 #include <stdexcept>
4 #include <string>
5 #include <fmt/format.h> // In C++20, this will be #include <format>
6 #include "Time.h" // include definition of class Time from Time.h
7 using namespace std;
8
9 // Time constructor initializes each data member
10 Time::Time(int hour, int minute, int second) {
11     setHour(hour); // validate and set private field m_hour
12     setMinute(minute); // validate and set private field m_minute
13     setSecond(second); // validate and set private field m_second
14 }
15
16 // set new Time value using 24-hour time
17 void Time::setTime(int hour, int minute, int second) {
18     // validate hour, minute and second
19     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
20         (second < 0 || second >= 60)) {
21         throw invalid_argument{"hour, minute or second was out of range"};
22     }
23
24     m_hour = hour;
25     m_minute = minute;
26     m_second = second;
27 }
28
29 // set hour value
30 void Time::setHour(int hour) {
31     if (hour < 0 || hour >= 24) {
32         throw invalid_argument{"hour must be 0-23"};
33     }
34
35     m_hour = hour;
36 }
```

```
37 // set minute value
38 void Time::setMinute(int minute) {
39     if (minute < 0 || minute >= 60) {
40         throw invalid_argument{"minute must be 0-59"};
41     }
42
43     m_minute = minute;
44 }
45
46 // set second value
47 void Time::setSecond(int second) {
48     if (second < 0 && second >= 60) {
49         throw invalid_argument{"second must be 0-59"};
50     }
51
52     m_second = second;
53 }
54
55
56 // return hour value
57 int Time::getHour() const {return m_hour;}
58
59 // return minute value
60 int Time::getMinute() const {return m_minute;}
61
62 // return second value
63 int Time::getSecond() const {return m_second;}
64
65 // return Time as a string in 24-hour format (HH:MM:SS)
66 string Time::to24HourString() const {
67     return fmt::format("{:02d}:{:02d}:{:02d}",
68                         getHour(), getMinute(), getSecond());
69 }
70
71 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
72 string Time::to12HourString() const {
73     return fmt::format("{}:{}{:02d} {}",
74                         ((getHour() % 12 == 0) ? 12 : getHour() % 12),
75                         getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
76 }
```

```
1 // fig09_12.cpp
2 // Constructor with default arguments.
3 #include <iostream>
4 #include <stdexcept>
5 #include <string>
6 #include <fmt/format.h> // In C++20, this will be #include <format>
7 #include "Time.h" // include definition of class Time from Time.h
8 using namespace std;
9
10 // displays a Time in 24-hour and 12-hour formats
11 void displayTime(string_view message, const Time& time) {
12     cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}"
13                         "\n", message, time.to24HourString(), time.to12HourString());
14 }
15
16 int main() {
17     const Time t1{}; // all arguments defaulted
18     const Time t2{2}; // hour specified; minute and second defaulted
19     const Time t3{21, 34}; // hour and minute specified; second defaulted
20     const Time t4{12, 25, 42}; // hour, minute and second specified
21
22     cout << "Constructed with:\n\n";
23     displayTime("t1: all arguments defaulted", t1);
24     displayTime("t2: hour specified; minute and second defaulted", t2);
25     displayTime("t3: hour and minute specified; second defaulted", t3);
26     displayTime("t4: hour, minute and second specified", t4);
27 }
```

```
28 // attempt to initialize t5 with invalid values
29 try {
30     const Time t5{27, 74, 99}; // all bad values specified
31 }
32 catch (const invalid_argument& e) {
33     cerr << fmt::format("t5 not created: {}\\n", e.what());
34 }
35 }
```

Constructed with:

t1: all arguments defaulted
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

t2: hour specified; minute and second defaulted
24-hour time: 02:00:00
12-hour time: 2:00:00 AM

t3: hour and minute specified; second defaulted
24-hour time: 21:34:00
12-hour time: 9:34:00 PM

t4: hour, minute and second specified
24-hour time: 12:25:42
12-hour time: 12:25:42 PM

t5 not created: hour must be 0-23

```
Time(); // default m_hour, m_minute and m_second to 0
explicit Time(int hour); // default m_minute & m_second to 0
Time(int hour, int minute); // default m_second to 0
Time(int hour, int minute, int second); // no default values
Time::Time() : Time{0, 0, 0} {}

Time::Time(int hour) : Time{hour, 0, 0} {}

Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```

```
1 // Fig. 9.13: CreateAndDestroy.h
2 // CreateAndDestroy class definition.
3 // Member functions defined in CreateAndDestroy.cpp.
4 #pragma once // prevent multiple inclusions of header
5 #include <string>
6 #include <string_view>
7
8 class CreateAndDestroy {
9 public:
10     CreateAndDestroy(int ID, std::string_view message); // constructor
11     ~CreateAndDestroy(); // destructor
12 private:
13     int m_ID; // ID number for object
14     std::string m_message; // message describing object
15 };
```

```
1 // Fig. 9.14: CreateAndDestroy.cpp
2 // CreateAndDestroy class member-function definitions.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
6 using namespace std;
7
8 // constructor sets object's ID number and descriptive message
9 CreateAndDestroy::CreateAndDestroy(int ID, string_view message)
10    : m_ID{ID}, m_message{message} {
11        cout << fmt::format("Object {}  constructor runs  {}\n",
12                           m_ID, m_message);
13    }
14
15 // destructor
16 CreateAndDestroy::~CreateAndDestroy() {
17     // output newline for certain objects; helps readability
18     cout << fmt::format("{}Object {}  destructor runs  {}\n",
19                         (m_ID == 1 || m_ID == 6 ? "\n" : ""), m_ID, m_message);
20 }
```

```
1 // fig09_15.cpp
2 // Order in which constructors and
3 // destructors are called.
4 #include <iostream>
5 #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
6 using namespace std;
7
8 void create(); // prototype
9
10 const CreateAndDestroy first{1, "(global before main)"};
11
12 int main() {
13     cout << "\nMAIN FUNCTION: EXECUTION BEGINS\n";
14     const CreateAndDestroy second{2, "(local in main)"};
15     static const CreateAndDestroy third{3, "(local static in main)"};
16
17     create(); // call function to create objects
18
19     cout << "\nMAIN FUNCTION: EXECUTION RESUMES\n";
20     const CreateAndDestroy fourth{4, "(local in main)"};
21     cout << "\nMAIN FUNCTION: EXECUTION ENDS\n";
22 }
23
24 // function to create objects
25 void create() {
26     cout << "\nCREATE FUNCTION: EXECUTION BEGINS\n";
27     const CreateAndDestroy fifth{5, "(local in create)"};
28     static const CreateAndDestroy sixth{6, "(local static in create)"};
29     const CreateAndDestroy seventh{7, "(local in create)"};
30     cout << "\nCREATE FUNCTION: EXECUTION ENDS\n";
31 }
```

Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS

Object 2 constructor runs (local in main)

Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS

Object 5 constructor runs (local in create)

Object 6 constructor runs (local static in create)

Object 7 constructor runs (local in create)

CREATE FUNCTION: EXECUTION ENDS

Object 7 destructor runs (local in create)

Object 5 destructor runs (local in create)

MAIN FUNCTION: EXECUTION RESUMES

Object 4 constructor runs (local in main)

MAIN FUNCTION: EXECUTION ENDS

Object 4 destructor runs (local in main)

Object 2 destructor runs (local in main)

Object 6 destructor runs (local static in create)

Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)

```
1 // Fig. 9.16: Time.h
2 // Time class definition.
3 // Member functions defined in Time.cpp
4
5 // prevent multiple inclusions of header
6 #pragma once
7
8 class Time {
9 public:
10    void setTime(int hour, int minute, int second);
11    int getHour() const;
12    int& badSetHour(int h); // dangerous reference return
13 private:
14    int m_hour{0};
15    int m_minute{0};
16    int m_second{0};
17};
```

```
1 // Fig. 9.17: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept>
4 #include "Time.h" // include definition of class Time
5 using namespace std;
6
7 // set values of hour, minute and second
8 void Time::setTime(int hour, int minute, int second) {
9     // validate hour, minute and second
10    if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
11        (second < 0 || second >= 60)) {
12        throw invalid_argument{"hour, minute or second was out of range"};
13    }
14
15    m_hour = hour;
16    m_minute = minute;
17    m_second = second;
18}
19
```

```
20 // return hour value
21 int Time::getHour() const {return m_hour;}
22
23 // poor practice: returning a reference to a private data member.
24 int& Time::badSetHour(int hour) {
25     if (hour < 0 || hour >= 24) {
26         throw invalid_argument{"hour must be 0-23"};
27     }
28
29     m_hour = hour;
30     return m_hour; // dangerous reference return
31 }
```

```
1 // fig09_18.cpp
2 // public member function that
3 // returns a reference to private data.
4 #include <iostream>
5 #include <fmt/format.h>
6 #include "Time.h" // include definition of class Time
7 using namespace std;
8
9 int main() {
10     Time t{}; // create Time object
11
12     // initialize hourRef with the reference returned by badSetHour
13     int& hourRef{t.badSetHour(20)}; // 20 is a valid hour
14
15     cout << fmt::format("Valid hour before modification: {}\n", hourRef);
16     hourRef = 30; // use hourRef to set invalid value in Time object t
17     cout << fmt::format("Invalid hour after modification: {}\n\n",
18                         t.getHour());
19
20     // Dangerous: Function call that returns a reference can be
21     // used as an lvalue! POOR PROGRAMMING PRACTICE!!!!!!!
22     t.badSetHour(12) = 74; // assign another invalid value to hour
23
24     cout << "After using t.badSetHour(12) as an lvalue, "
25         << fmt::format("hour is: {}\n", t.getHour());
26 }
```

```
Valid hour before modification: 20
Invalid hour after modification: 30
After using t.badSetHour(12) as an lvalue, hour is: 74
```

```
1 // Fig. 9.19: Date.h
2 // Date class declaration. Member functions are defined in Date.cpp.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5
6 // class Date definition
7 class Date {
8 public:
9     explicit Date(int year, int month, int day);
10    std::string toString() const;
11 private:
12    int m_year;
13    int m_month;
14    int m_day;
15};
```

```
1 // Fig. 9.20: Date.cpp
2 // Date class member-function definitions.
3 #include <string>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Date.h" // include definition of class Date from Date.h
6 using namespace std;
7
8 // Date constructor (should do range checking)
9 Date::Date(int year, int month, int day)
10   : m_year{year}, m_month{month}, m_day{day} {}
11
12 // return string representation of a Date in the format yyyy-mm-dd
13 string Date::toString() const {
14     return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
15 }
```

```
1 // fig09_21.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using the default assignment operator.
4 #include <iostream>
5 #include <fmt/format.h> // In C++20, this will be #include <format>
6 #include "Date.h" // include definition of class Date from Date.h
7 using namespace std;
8
9 int main() {
10     const Date date1{2004, 7, 4};
11     Date date2{2020, 1, 1};
12
13     cout << fmt::format("date1: {}\ndate2: {}\n\n",
14                         date1.toString(), date2.toString());
15     date2 = date1; // uses the default assignment operator
16     cout << fmt::format("After assignment, date2: {}\n", date2.toString());
17 }
```

```
date1: 2004-07-04
date2: 2020-01-01
```

```
After assignment, date2: 2004-07-04
```

```
const Time noon{12, 0, 0};
```

```
1 // fig09_22.cpp
2 // const objects and const member functions.
3 #include "Time.h" // include Time class definition
4
5 int main() {
6     Time wakeUp{6, 45, 0}; // non-constant object
7     const Time noon{12, 0, 0}; // constant object
8
9         // OBJECT      MEMBER FUNCTION
10    wakeUp.setHour(18);    // non-const  non-const
11    noon.setHour(12);    // const      non-const
12    wakeUp.getHour();    // non-const  const
13    noon.getMinute();    // const      const
14    noon.to24HourString(); // const      const
15    noon.to12HourString(); // const      non-const
16 }
```

Microsoft Visual C++ compiler error messages:

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\fig09_22.cpp(11,19):
error C2662: 'void Time::setHour(int)': cannot convert 'this' pointer from
'const Time' to 'Time &'
```

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\fig09_22.cpp(11,4):
message : Conversion loses qualifiers
```

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\Time.h(15,9):
message : see declaration of 'Time::setHour'
```

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\fig09_22.cpp(15,26):
error C2662: 'std::string Time::to12HourString(void)': cannot convert 'this'
pointer from 'const Time' to 'Time &'
```

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\fig09_22.cpp(15,4):
message : Conversion loses qualifiers
```

```
C:\Users\PaulDeitel\Documents\examples\ch09\fig09_22\Time.h(25,16):
message : see declaration of 'Time::to12HourString'
```

```
1 // Fig. 9.23: Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #pragma once // prevent multiple inclusions of header
4 #include <iostream>
5
6 class Date {
7 public:
8     static const int monthsPerYear{12}; // months in a year
9     explicit Date(int year, int month, int day);
10    std::string toString() const; // date string in yyyy-mm-dd format
11    ~Date(); // provided to show when destruction occurs
12 private:
13     int m_year; // any year
14     int m_month; // 1-12 (January-December)
15     int m_day; // 1-31 based on month
16
17     // utility function to check if day is proper for month and year
18     bool checkDay(int day) const;
19 };
```

```
1 // Fig. 9.24: Date.cpp
2 // Date class member-function definitions.
3 #include <iostream>
4 #include <array>
5 #include <stdexcept>
6 #include <fmt/format.h> // In C++20, this will be #include <format>
7 #include "Date.h" // include Date class definition
8 using namespace std;
9
10 // constructor confirms proper value for month; calls
11 // utility function checkDay to confirm proper value for day
12 Date::Date(int year, int month, int day)
13     : m_year{year}, m_month{month}, m_day{day} {
14     if (m_month < 1 || m_month > monthsPerYear) { // validate the month
15         throw invalid_argument{"month must be 1-12"};
16     }
17 }
```

```
18     if (!checkDay(day)) { // validate the day
19         throw invalid_argument{"Invalid day for current month and year"};
20     }
21
22     // output Date object to show when its constructor is called
23     cout << fmt::format("Date object constructor: {}\n", toString());
24 }
25
26 // gets string representation of a Date in the form yyyy-mm-dd
27 string Date::toString() const {
28     return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
29 }
30
31 // output Date object to show when its destructor is called
32 Date::~Date() {
33     cout << fmt::format("Date object destructor: {}\n", toString());
34 }
35
36 // utility function to confirm proper day value based on
37 // month and year; handles leap years, too
38 bool Date::checkDay(int day) const {
39     static const array<int> daysPerMonth{
40         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
41
42     // determine whether testDay is valid for specified month
43     if (1 <= day && day <= daysPerMonth.at(m_month)) {
44         return true;
45     }
46
47     // February 29 check for leap year
48     if (m_month == 2 && day == 29 && (m_year % 400 == 0 ||
49         (m_year % 4 == 0 && m_year % 100 != 0))) {
50         return true;
51     }
52
53     return false; // invalid day, based on current m_month and m_year
54 }
```

```
1 // Fig. 9.25: Employee.h
2 // Employee class definition showing composition.
3 // Member functions defined in Employee.cpp.
4 #pragma once // prevent multiple inclusions of header
5 #include <string>
6 #include <string_view>
7 #include "Date.h" // include Date class definition
8
9 class Employee {
10 public:
11     Employee(std::string_view firstName, std::string_view lastName,
12             const Date& birthDate, const Date& hireDate);
13     std::string toString() const;
14     ~Employee(); // provided to confirm destruction order
15 private:
16     std::string m(firstName); // composition: member object
17     std::string m(lastName); // composition: member object
18     Date m(birthDate); // composition: member object
19     Date m(hireDate); // composition: member object
20 };
```

```
1 // fig09_27.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Date.h" // Date class definition
6 #include "Employee.h" // Employee class definition
7 using namespace std;
8
9 int main() {
10     const Date birth{1987, 7, 24};
11     const Date hire{2018, 3, 12};
12     const Employee manager{"Sue", "Green", birth, hire};
13
14     cout << fmt::format("\n{}\n", manager.toString());
15 }
```

```
Date object constructor: 1987-07-24
Date object constructor: 2018-03-12
Employee object constructor: Sue Green
```

```
Green, Sue Hired: 2018-03-12 Birthday: 1987-07-24
Employee object destructor: Green, Sue
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
```

```
1 // fig09_28.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 using namespace std;
```

```
6
7 // Count class definition
8 class Count {
9     friend void setX(Count& c, int value); // friend declaration
10    public:
11        int getX() const {return m_x;}
12    private:
13        int m_x{0};
14    };
15
16 // function setX can modify private data of Count
17 // because setX is declared as a friend of Count (line 8)
18 void setX(Count& c, int value) {
19     c.m_x = value; // allowed because setX is a friend of Count
20 }
21
22 int main() {
23     Count counter{}; // create Count object
24
25     cout << fmt::format("Initial counter.x value: {}\n", counter.getX());
26     setX(counter, 8); // set x using a friend function
27     cout << fmt::format("counter.x after setX call: {}\n", counter.getX());
28 }
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

```
// set hour value
void Time::setHour(int hour) {
    if (hour < 0 || hour >= 24) {
        throw invalid_argument{"hour must be 0-23"};
    }

    this->hour = hour; // use this-> to access data member
}
```

```
1 // fig09_29.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 using namespace std;
6
```

```
7 class Test {
8 public:
9     explicit Test(int value);
10    void print() const;
11 private:
12     int m_x{0};
13 };
14
15 // constructor
16 Test::Test(int value) : m_x{value} {} // initialize x to value
17
18 // print x using implicit then explicit this pointers;
19 // the parentheses around *this are required due to precedence
20 void Test::print() const {
21     // implicitly use the this pointer to access the member x
22     cout << fmt::format("      x = {}\n", m_x);
23
24     // explicitly use the this pointer and the arrow operator
25     // to access the member x
26     cout << fmt::format(" this->x = {}\n", this->m_x);
27
28     // explicitly use the dereferenced this pointer and
29     // the dot operator to access the member x
30     cout << fmt::format("(*this).x = {}\n", (*this).m_x);
31 }
32
33 int main() {
34     const Test testObject{12}; // instantiate and initialize testObject
35     testObject.print();
36 }
```

```
x = 12
this->x = 12
(*this).x = 12
```

```
1 // Fig. 9.30: Time.h
2 // Time class modified to enable cascaded member-function calls.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5
6 class Time {
7 public:
8     // default constructor because it can be called with no arguments
9     explicit Time(int hour = 0, int minute = 0, int second = 0);
10
11    // set functions
12    Time& setTime(int hour, int minute, int second);
13    Time& setHour(int hour); // set hour (after validation)
14    Time& setMinute(int minute); // set minute (after validation)
15    Time& setSecond(int second); // set second (after validation)
16
17    int getHour() const; // return hour
18    int getMinute() const; // return minute
19    int getSecond() const; // return second
20    std::string to24HourString() const; // 24-hour time format string
21    std::string to12HourString() const; // 12-hour time format string
22 private:
23     int m_hour{0}; // 0 - 23 (24-hour clock format)
24     int m_minute{0}; // 0 - 59
25     int m_second{0}; // 0 - 59
26 };
```

```
1 // Fig. 9.31: Time.cpp
2 // Time class member-function definitions.
3 #include <stdexcept>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Time.h" // Time class definition
6 using namespace std;
7
8 // Time constructor initializes each data member
9 Time::Time(int hour, int minute, int second) {
10     setHour(hour); // validate and set private field m_hour
11     setMinute(minute); // validate and set private field m_minute
12     setSecond(second); // validate and set private field m_second
13 }
14
```

```
15 // set new Time value using 24-hour time
16 Time& Time::setTime(int hour, int minute, int second) {
17     Time time{hour, minute, second}; // create a temporary Time object
18     *this = time; // if time is valid, assign its members to current object
19     return *this; // enables cascading
20 }
21
22 // set hour value
23 Time& Time::setHour(int hour) { // note Time& return
24     if (hour < 0 || hour >= 24) {
25         throw invalid_argument{"hour must be 0-23"};
26     }
27
28     m_hour = hour;
29     return *this; // enables cascading
30 }
31
32 // set minute value
33 Time& Time::setMinute(int m) { // note Time& return
34     if (m < 0 || m >= 60) {
35         throw invalid_argument{"minute must be 0-59"};
36     }
37
38     m_minute = m;
39     return *this; // enables cascading
40 }
41
42 // set second value
43 Time& Time::setSecond(int s) { // note Time& return
44     if (s < 0 || s >= 60) {
45         throw invalid_argument{"second must be 0-59"};
46     }
47
48     m_second = s;
49     return *this; // enables cascading
50 }
51
52 // get hour value
53 int Time::getHour() const {return m_hour;}
54
55 // get minute value
56 int Time::getMinute() const {return m_minute;}
57
58 // get second value
59 int Time::getSecond() const {return m_second;}
60
61 // return Time as a string in 24-hour format (HH:MM:SS)
62 string Time::to24HourString() const {
63     return fmt::format("{:02d}:{:02d}:{:02d}",
64                         getHour(), getMinute(), getSecond());
65 }
```

```
66
67 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
68 string Time::to12HourString() const {
69     return fmt::format("{}:{}{:02d} {}",
70         ((getHour() % 12 == 0) ? 12 : getHour() % 12),
71         getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
72 }
```

```
1 // fig09_32.cpp
2 // Cascading member-function calls with the this pointer.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Time.h" // Time class definition
6 using namespace std;
7
8 int main() {
9     Time t{}; // create Time object
10
11     t.setHour(18).setMinute(30).setSecond(22); // cascaded function calls
12
13     // output time in 24-hour and 12-hour formats
14     cout << fmt::format("24-hour time: {}\n12-hour time: {}\n\n",
15                         t.to24HourString(), t.to12HourString());
16
17     // cascaded function calls
18     cout << fmt::format("New 12-hour time: {}\n",
19                         t.setTime(20, 20, 20).to12HourString());
20 }
```

```
24-hour time: 18:30:22
12-hour time: 6:30:22 PM
```

```
New 12-hour time: 8:20:20 PM
```

```
t.setHour(18).setMinute(30).setSecond(22);
t.setMinute(30).setSecond(22);
```

```
1 // Fig. 9.33: Employee.h
2 // Employee class definition with a static data member to
3 // track the number of Employee objects in memory
4 #pragma once
5 #include <string>
6 #include <string_view>
7
8 class Employee {
9 public:
10    Employee(std::string_view firstName, std::string_view lastName);
11    ~Employee(); // destructor
12    const std::string& getFirstName() const; // return first name
13    const std::string& getLastName() const; // return last name
14
15    // static member function
16    static int getCount(); // return # of objects instantiated
17 private:
18    std::string m(firstName);
19    std::string m(lastName);
20
21    // static data
22    inline static int m_count{0}; // number of objects instantiated
23};
```

Images

```
1 // fig09_35.cpp
2 // static data member tracking the number of objects of a class.
3 #include <iostream>
4 #include <fmt/format.h> // In C++20, this will be #include <format>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 int main() {
9     // no objects exist; use class name and scope resolution
10    // operator to access static member function getCount
11    cout << fmt::format("Initial employee count: {}\n",
12                      Employee::getCount()); // use class name
13
14    // the following scope creates and destroys
15    // Employee objects before main terminates
16    {
17        const Employee e1{"Susan", "Baker"};
18        const Employee e2{"Robert", "Jones"};
19
20        // two objects exist; call static member function getCount again
21        // using the class name and the scope resolution operator
22        cout << fmt::format("Employee count after creating objects: {}\n\n",
23                           Employee::getCount());
24
25        cout << fmt::format("Employee 1: {} {}\nEmployee 2: {} {}\n\n",
26                           e1.getFirstName(), e1.getLastName(),
27                           e2.getFirstName(), e2.getLastName());
28    }
29
30    // no objects exist, so call static member function getCount again
31    // using the class name and the scope resolution operator
32    cout << fmt::format("Employee count after objects are deleted: {}\n",
33                      Employee::getCount());
34 }
```

```
Initial employee count: 0
Employee constructor called for Susan Baker
Employee constructor called for Robert Jones
Employee count after creating objects: 2

Employee 1: Susan Baker
Employee 2: Robert Jones
~Employee() called for Robert Jones
~Employee() called for Susan Baker
Employee count after objects are deleted: 0
```

```
struct Record {  
    int account;  
    string first;  
    string last;  
    double balance;  
};  
Record record{100, "Brian", "Blue", 123.45};  
struct Record {  
    int account;  
    string first;  
    string last;  
    double balance{0.0};  
};  
Record record{0, "Brian", "Blue"};  
Record record{.first{"Sue"}, .last{"Green"}};  
>{"account": 100, "name": "Jones", "balance": 24.98}
```

```
1 // fig09_36.cpp  
2 // Serializing and deserializing objects with the cereal library.  
3 #include <iostream>  
4 #include <fstream>  
5 #include <vector>  
6 #include <fmt/format.h> // In C++20, this will be #include <format>  
7 #include <cereal/archives/json.hpp>  
8 #include <cereal/types/vector.hpp>  
9  
10 using namespace std;
```

```
11  
12 struct Record {  
13     int account{};  
14     string first{};  
15     string last{};  
16     double balance{};  
17 };  
18
```

```
19 // function template serialize is responsible for serializing and
20 // deserializing Record objects to/from the specified Archive
21 template <typename Archive>
22 void serialize(Archive& archive, Record& record) {
23     archive(cereal::make_nvp("account", record.account),
24             cereal::make_nvp("first", record.first),
25             cereal::make_nvp("last", record.last),
26             cereal::make_nvp("balance", record.balance));
27 }
28
```

```
cereal::make_nvp("account", record.account)
```

```
29 // display record at command line
30 void displayRecords(const vector<Record>& records) {
31     for (const auto& r : records) {
32         cout << fmt::format("{} {} {} {:.2f}\n",
33                             r.account, r.first, r.last, r.balance);
34     }
35 }
36
```

```
37 int main() {
38     vector<Record> records{
39         Record{100, "Brian", "Blue", 123.45},
40         Record{200, "Sue", "Green", 987.65}
41     };
42
43     cout << "Records to serialize:\n";
44     displayRecords(records);
45 }
```

```
Records to serialize:
100 Brian Blue 123.45
200 Sue Green 987.65
```

```
46 // serialize vector of Records to JSON and store in text file
47 if (ofstream output{"records.json"}) {
48     cereal::JSONOutputArchive archive{output};
49     archive(cereal::make_nvp("records", records)); // serialize records
50 }
51

52 // deserialize JSON from text file into vector of Records
53 if (ifstream input{"records.json"}) {
54     cereal::JSONInputArchive archive{input};
55     vector<Record> deserializedRecords{};
56     archive(deserializedRecords); // deserialize records
57     cout << "\nDeserialized records:\n";
58     displayRecords(deserializedRecords);
59 }
60 }
```

Deserialized records:

100 Brian Blue 123.45

200 Sue Green 987.65

```
1 // fig09_37.cpp
2 // Serializing and deserializing objects containing private data.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <string_view>
7 #include <vector>
8 #include <fmt/format.h> // In C++20, this will be #include <format>
9 #include <cereal/archives/json.hpp>
10 #include <cereal/types/vector.hpp>
11
12 using namespace std;
13
14 class Record {
15     // declare serialize as a friend for direct access to private data
16     template<typename Archive>
17         friend void serialize(Archive& archive, Record& record);
18
19 public:
20     // constructor
21     explicit Record(int account = 0, string_view first = "",
22                     string_view last = "", double balance = 0.0)
23         : m_account{account}, m_first{first},
24           m_last{last}, m_balance{balance} {}
25
26     // get member functions
27     int getAccount() const {return m_account;}
```

```
28     const string& getFirst() const {return m_first;}
29     const string& getLast() const {return m_last;}
30     double getBalance() const {return m_balance;}
31
32 private:
33     int m_account{};
34     string m_first{};
35     string m_last{};
36     double m_balance{};
37 };
38
39 // function template serialize is responsible for serializing and
40 // deserializing Record objects to/from the specified Archive
41 template <typename Archive>
42 void serialize(Archive& archive, Record& record) {
43     archive(cereal::make_nvp("account", record.m_account),
44             cereal::make_nvp("first", record.m_first),
45             cereal::make_nvp("last", record.m_last),
46             cereal::make_nvp("balance", record.m_balance));
47 }
48
49 // display record at command line
50 void displayRecords(const vector<Record>& records) {
51     for (auto& r : records) {
52         cout << fmt::format("{} {} {} {:.2f}\n", r.getAccount(),
53                             r.getFirst(), r.getLast(), r.getBalance());
54     }
55 }
56
57 int main() {
58     vector<Record> records{
59         Record{100, "Brian", "Blue", 123.45},
60         Record{200, "Sue", "Green", 987.65}
61     };
62
63     cout << "Records to serialize:\n";
64     displayRecords(records);
65
66     // serialize vector of Records to JSON and store in text file
67     if (ofstream output{"records2.json"}) {
68         cereal::JSONOutputArchive archive{output};
69         archive(cereal::make_nvp("records", records)); // serialize records
70     }
71
72     // deserialize JSON from text file into vector of Records
73     if (ifstream input{"records2.json"}) {
74         cereal::JSONInputArchive archive{input};
75         vector<Record> serializedRecords{};
76         archive(serializedRecords); // deserialize records
77         cout << "\nDeserialized records:\n";
78         displayRecords(serializedRecords);
79     }
80 }
```

```
class TwoDimensionalShape : public Shape
```

```
1 // Fig. 10.1: SalariedEmployee.h
2 // SalariedEmployee class definition.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5 #include <string_view>
6
7 class SalariedEmployee {
8 public:
9     SalariedEmployee(std::string_view name, double salary);
10
11    void setName(std::string_view name);
12    std::string getName() const;
13
14    void setSalary(double salary);
15    double getSalary() const;
16
17    double earnings() const;
18    std::string toString() const;
19 private:
20     std::string m_name{};
21     double m_salary{0.0};
22 };
```

```
1 // Fig. 10.2: SalariedEmployee.cpp
2 // Class SalariedEmployee member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "SalariedEmployee.h" // SalariedEmployee class definition
6 using namespace std;
7
8 // constructor
9 SalariedEmployee::SalariedEmployee(string_view name, double salary)
10    : m_name{name} {
11    setSalary(salary);
12 }
13
14 // set name
15 void SalariedEmployee::setName(string_view name) {
16    m_name = name; // should validate
17 }
18
19 // return name
20 string SalariedEmployee::getName() const {return m_name;}
21
22 // set salary
23 void SalariedEmployee::setSalary(double salary) {
24    if (salary < 0.0) {
25        throw invalid_argument("Salary must be >= 0.0");
26    }
27
28    m_salary = salary;
29 }
30
31 // return salary
32 double SalariedEmployee::getSalary() const {return m_salary;}
33
34 // calculate earnings
35 double SalariedEmployee::earnings() const {return getSalary();}
36
37 // return string representation of SalariedEmployee object
38 string SalariedEmployee::toString() const {
39    return fmt::format("name: {}\\nsalary: ${:.2f}\\n", getName(),
40                      getSalary());
41 }
```

```
1 // fig10_03.cpp
2 // SalariedEmployee class test program.
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "SalariedEmployee.h" // SalariedEmployee class definition
6 using namespace std;
7
8 int main() {
9     // instantiate a SalariedEmployee object
10    SalariedEmployee employee{"Sue Jones", 300.0};
11
12    // get SalariedEmployee data
13    cout << "Employee information obtained by get functions:\n"
14        << fmt::format("name: {}\nsalary: ${:.2f}\n", employee.getName(),
15                      employee.getSalary());
16
17    employee.setSalary(500.0); // change salary
18    cout << "\nUpdated employee information from function toString:\n"
19        << employee.toString();
20
21    // display only the employee's earnings
22    cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
23 }
```

Employee information obtained by get functions:
name: Sue Jones
salary: \$300.00

Updated employee information from function toString:
name: Sue Jones
salary: \$500.00

earnings: \$500.00

```
1 // Fig. 10.4: SalariedCommissionEmployee.h
2 // SalariedCommissionEmployee class derived from class SalariedEmployee.
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "SalariedEmployee.h"
7
8 class SalariedCommissionEmployee : public SalariedEmployee {
9 public:
10     SalariedCommissionEmployee(std::string_view name, double salary,
11                               double grossSales, double commissionRate);
12
13     void setGrossSales(double grossSales);
14     double getGrossSales() const;
15
16     void setCommissionRate(double commissionRate);
17     double getCommissionRate() const;
18
19     double earnings() const;
20     std::string toString() const;
21 private:
22     double m_grossSales{0.0};
23     double m_commissionRate{0.0};
24 };
```

```
1 // Fig. 10.5: SalariedCommissionEmployee.cpp
2 // Class SalariedCommissionEmployee member-function definitions.
3 #include "fmt/format.h" // In C++20, this will be #include <format>
4 #include <stdexcept>
5 #include "SalariedCommissionEmployee.h"
6 using namespace std;
7
8 // constructor
9 SalariedCommissionEmployee::SalariedCommissionEmployee(string_view name,
10     double salary, double grossSales, double commissionRate)
11     : SalariedEmployee{name, salary} { // call base-class constructor
12
13     setGrossSales(grossSales); // validate & store gross sales
14     setCommissionRate(commissionRate); // validate & store commission rate
15 }
16
17 // set gross sales amount
18 void SalariedCommissionEmployee::setGrossSales(double grossSales) {
19     if (grossSales < 0.0) {
20         throw invalid_argument("Gross sales must be >= 0.0");
21     }
22
23     m_grossSales = grossSales;
24 }
25
```

```
26 // return gross sales amount
27 double SalariedCommissionEmployee::getGrossSales() const {
28     return m_grossSales;
29 }
30
31 // set commission rate
32 void SalariedCommissionEmployee::setCommissionRate(
33     double commissionRate) {
34
35     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
36         throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
37     }
38
39     m_commissionRate = commissionRate;
40 }
41
42 // get commission rate
43 double SalariedCommissionEmployee::getCommissionRate() const {
44     return m_commissionRate;
45 }
46
47 // calculate earnings--uses SalariedEmployee::earnings()
48 double SalariedCommissionEmployee::earnings() const {
49     return SalariedEmployee::earnings() +
50         getGrossSales() * getCommissionRate();
51 }
52
53 // returns string representation of SalariedCommissionEmployee object
54 string SalariedCommissionEmployee::toString() const {
55     return fmt::format(
56         "{}gross sales: ${:.2f}\ncommission rate: {:.2f}\n",
57         SalariedEmployee::toString(), getGrossSales(), getCommissionRate());
58 }
```

```
SalariedEmployee::earnings()
```

```
1 // fig10_06.cpp
2 // SalariedCommissionEmployee class test program.
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "SalariedCommissionEmployee.h"
6 using namespace std;
7
8 int main() {
9     // instantiate SalariedCommissionEmployee object
10    SalariedCommissionEmployee employee{"Bob Lewis", 300.0, 5000.0, .04};
11
12    // get SalariedCommissionEmployee data
13    cout << "Employee information obtained by get functions:\n"
14    << fmt::format("{}: {}\n{}: {:.2f}\n{}: {:.2f}\n{}: {:.2f}\n",
15                  "name", employee.getName(), "salary", employee.getSalary(),
16                  "gross sales", employee.getGrossSales(),
17                  "commission", employee.getCommissionRate());
18
19    employee.setGrossSales(8000.0); // change gross sales
20    employee.setCommissionRate(0.1); // change commission rate
21    cout << "\nUpdated employee information from function toString:\n"
22    << employee.toString();
23
24    // display the employee's earnings
25    cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
26 }
```

Employee information obtained by get functions:

name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission: 0.04

Updated employee information from function toString:

name: Bob Lewis
salary: \$300.00
gross sales: \$8000.00
commission rate: 0.10

earnings: \$1100.00

'double SalariedEmployee::m_salary' is private within this context

```
1 // fig10_07.cpp
2 // Aiming base-class and derived-class pointers at base-class
3 // and derived-class objects, respectively.
4 #include <iostream>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 #include "SalariedEmployee.h"
7 #include "SalariedCommissionEmployee.h"
8 using namespace std;
9
10 int main() {
11     // create base-class object
12     SalariedEmployee salaried{"Sue Jones", 500.0};
13
14     // create derived-class object
15     SalariedCommissionEmployee salariedCommission{
16         "Bob Lewis", 300.0, 5000.0, .04};
17
18     // output objects salaried and salariedCommission
19     cout << fmt::format("{}:\n{}\n{}:\n{}",
20                         "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS",
21                         salaried.toString(), // base-class toString
22                         salariedCommission.toString()); // derived-class toString
23
24     // natural: aim base-class pointer at base-class object
25     SalariedEmployee* salariedPtr=&salaried;
26     cout << fmt::format("{}:\n{}:\n{}:\n{}",
27                         "CALLING TOSTRING WITH BASE-CLASS POINTER TO",
28                         "BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
29                         salariedPtr->toString()); // base-class version
30 }
```

```
31 // natural: aim derived-class pointer at derived-class object
32 SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
33
34 cout << fmt::format("{}\n{}:\n{}\n",
35                     "CALLING TOSTRING WITH DERIVED-CLASS POINTER TO",
36                     "DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
37                     salariedCommissionPtr->toString()); // derived-class version
38
39 // aim base-class pointer at derived-class object
40 salariedPtr = &salariedCommission;
41 cout << fmt::format("{}\n{}:\n{}\n",
42                     "CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS",
43                     "OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
44                     salariedPtr->toString()); // base version
45 }
```

DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:

name: Sue Jones

salary: \$500.00

name: Bob Lewis

salary: \$300.00

gross sales: \$5000.00

commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:

name: Sue Jones

salary: \$500.00

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO

DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:

name: Bob Lewis

salary: \$300.00

gross sales: \$5000.00

commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS
OBJECT INVOKES BASE-CLASS FUNCTIONALITY:

name: Bob Lewis

salary: \$300.00

```
1 // fig10_08.cpp
2 // Aiming a derived-class pointer at a base-class object.
3 #include "SalariedEmployee.h"
4 #include "SalariedCommissionEmployee.h"
5
6 int main() {
7     SalariedEmployee salaried{"Sue Jones", 500.0};
8
9     // aim derived-class pointer at base-class object
10    // Error: a SalariedEmployee is not a SalariedCommissionEmployee
11    SalariedCommissionEmployee* salariedCommissionPtr{&salaried};
12 }
```

Microsoft Visual C++ compiler error message:

```
fig10_08.cpp(11,63): error C2440: 'initializing': cannot convert from
'SalariedEmployee *' to 'SalariedCommissionEmployee *'
```

```
1 // fig10_09.cpp
2 // Attempting to call derived-class-only functions
3 // via a base-class pointer.
4 #include <string>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7 using namespace std;
8
9 int main() {
10     SalariedCommissionEmployee salariedCommission{
11         "Bob Lewis", 300.0, 5000.0, .04};
12
13     // aim base-class pointer at derived-class object (allowed)
14     SalariedEmployee* salariedPtr{&salariedCommission};
15
16     // invoke base-class member functions on derived-class
17     // object through base-class pointer (allowed)
18     string name{salariedPtr->getName()};
19     double salary{salariedPtr->getSalary()};
20
21     // attempt to invoke derived-class-only member functions
22     // on derived-class object through base-class pointer (disallowed)
23     double grossSales{salariedPtr->getGrossSales()};
24     double commissionRate{salariedPtr->getCommissionRate()};
25     salariedPtr->setGrossSales(8000.0);
26 }
```

GNU C++ compiler error messages:

```
fig10_09.cpp: In function ‘int main()’:
fig10_09.cpp:23:35: error: ‘class SalariedEmployee’ has no member named
‘getGrossSales’
23 |     double grossSales{salariedPtr->getGrossSales()};
|           ^
fig10_09.cpp:24:39: error: ‘class SalariedEmployee’ has no member named
‘getCommissionRate’
24 |     double commissionRate{salariedPtr->getCommissionRate()};
|           ^
fig10_09.cpp:25:17: error: ‘class SalariedEmployee’ has no member named
‘setGrossSales’
25 |     salariedPtr->setGrossSales(8000.0);
|           ^
```

```
double earnings() const;  
std::string toString() const;
```

```
virtual double earnings() const;  
virtual std::string toString() const;
```

```
double earnings() const;  
std::string toString() const;
```

```
double earnings() const override;  
std::string toString() const override;
```

```
1 // fig10_10.cpp
2 // Introducing polymorphism, virtual functions and dynamic binding.
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7 using namespace std;
8
9 int main() {
10     // create base-class object
11     SalariedEmployee salaried{"Sue Jones", 500.0};
12
13     // create derived-class object
14     SalariedCommissionEmployee salariedCommission{
15         "Bob Lewis", 300.0, 5000.0, .04};
16
17     // output objects using static binding
18     cout << fmt::format("{}\n{}:\n{}\n{}\n",
19                         "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND",
20                         "DERIVED-CLASS OBJECTS WITH STATIC BINDING",
21                         salaried.toString(), // static binding
22                         salariedCommission.toString()); // static binding
23
24     cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND\n"
25         << "DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING\n\n";
26 }
```

```
27 // natural: aim base-class pointer at base-class object
28 SalariedEmployee* salariedPtr{&salaried};
29 cout << fmt::format("{}\n{}:\n{}\n",
30                     "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
31                     "TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
32                     salariedPtr->toString()); // base-class version
33
34 // natural: aim derived-class pointer at derived-class object
35 SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
36 cout << fmt::format("{}\n{}{}:\n{}\n",
37                     "CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS",
38                     "POINTER TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS ",
39                     "FUNCTIONALITY",
40                     salariedCommissionPtr->toString()); // derived-class version
41
42 // aim base-class pointer at derived-class object
43 salariedPtr = &salariedCommission;
44
45 // runtime polymorphism: invokes SalariedCommissionEmployee
46 // via base-class pointer to derived-class object
47 cout << fmt::format("{}\n{}{}:\n{}\n",
48                     "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
49                     "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS ",
50                     "FUNCTIONALITY",
51                     salariedPtr->toString()); // derived-class version
52 }
```

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH STATIC BINDING:

name: Sue Jones
salary: \$500.00

name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:

name: Sue Jones
salary: \$500.00

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS
POINTER TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:

name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:

name: Bob Lewis
salary: \$300.00
gross sales: \$5000.00
commission rate: 0.04

```
virtual ~SalariedEmployee() = default;
```

```
class DerivedClass : public BaseClass final {  
    // class body  
};
```

```
virtual void draw() const = 0; // pure virtual function
```

```
1 // Fig. 10.11: Employee.h
2 // Employee abstract base class.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5 #include <string_view>
6
7 class Employee {
8 public:
9     explicit Employee(std::string_view name);
10    virtual ~Employee() = default; // compiler generates virtual destructor
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    // pure virtual function makes Employee an abstract base class
16    virtual double earnings() const = 0; // pure virtual
17    virtual std::string toString() const; // virtual
18 private:
19     std::string m_name;
20};
```

```
1 // Fig. 10.12: Employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // constructor
9 Employee::Employee(string_view name) : m_name{name} {} // empty body
10
11 // set name
12 void Employee::setName(string_view name) {m_name = name;}
13
14 // get name
15 string Employee::getName() const {return m_name;}
16
17 // return string representation of an Employee
18 string Employee::toString() const {
19     return fmt::format("name: {}", getName());
20 }
```

```
1 // Fig. 10.13: SalariedEmployee.h
2 // SalariedEmployee class derived from Employee.
3 #pragma once
4 #include <string> // C++ standard string class
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class SalariedEmployee final : public Employee {
9 public:
10     SalariedEmployee(std::string_view name, double salary);
11     virtual ~SalariedEmployee() = default; // virtual destructor
12
13     void setSalary(double salary);
14     double getSalary() const;
15
16     // keyword override signals intent to override
17     double earnings() const override; // calculate earnings
18     std::string toString() const override; // string representation
19 private:
20     double m_salary{0.0};
21 };
```

```
1 // Fig. 10.15: CommissionEmployee.h
2 // CommissionEmployee class derived from Employee.
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class CommissionEmployee final : public Employee {
9 public:
10     CommissionEmployee(std::string_view name, double grossSales,
11                         double commissionRate);
12     virtual ~CommissionEmployee() = default; // virtual destructor
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19
20     // keyword override signals intent to override
21     double earnings() const override; // calculate earnings
22     std::string toString() const override; // string representation
23 private:
24     double m_grossSales{0.0};
25     double m_commissionRate{0.0};
26 }
```

```
1 // Fig. 10.16: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "CommissionEmployee.h" // CommissionEmployee class definition
6 using namespace std;
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(string_view name,
10     double grossSales, double commissionRate) : Employee{name} {
11     setGrossSales(grossSales);
12     setCommissionRate(commissionRate);
13 }
14
15 // set gross sales amount
16 void CommissionEmployee::setGrossSales(double grossSales) {
17     if (grossSales < 0.0) {
18         throw invalid_argument("Gross sales must be >= 0.0");
19     }
20
21     m_grossSales = grossSales;
22 }
23
24 // return gross sales amount
25 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
26
27 // set commission rate
28 void CommissionEmployee::setCommissionRate(double commissionRate) {
29     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
30         throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
31     }
32
33     m_commissionRate = commissionRate;
34 }
35
36 // return commission rate
37 double CommissionEmployee::getCommissionRate() const {
38     return m_commissionRate;
39 }
40
41 // calculate earnings
42 double CommissionEmployee::earnings() const {
43     return getGrossSales() * getCommissionRate();
44 }
45
46 // return string representation of CommissionEmployee object
47 string CommissionEmployee::toString() const {
48     return fmt::format("{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::toString(),
49                       "gross sales", getGrossSales(),
50                       "commission rate", getCommissionRate());
51 }
```

```
1 // fig10_17.cpp
2 // Processing Employee derived-class objects with variable-name handles
3 // then polymorphically using base-class pointers and references
4 #include <iostream>
5 #include <vector>
6 #include "fmt/format.h" // In C++20, this will be #include <format>
7 #include "Employee.h"
8 #include "SalariedEmployee.h"
9 #include "CommissionEmployee.h"
10 using namespace std;
11
12 void virtualViaPointer(const Employee* baseClassPtr); // prototype
13 void virtualViaReference(const Employee& baseClassRef); // prototype
14
15 int main() {
16     // create derived-class objects
17     SalariedEmployee salaried{"John Smith", 800.0};
18     CommissionEmployee commission{"Sue Jones", 10000, .06};
19
20     // output each Employee
21     cout << "EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING\n"
22         << fmt::format("{}\n{}{:2f}\n{}\n{}{:2f}\n{}\n",
23             salaried.toString(), "earned $", salaried.earnings(),
24             commission.toString(), "earned $", commission.earnings());
25
26     // create and initialize vector of base-class pointers
27     vector<Employee*> employees{&salaried, &commission};
28
29     cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING\n\n";
```

```
31 // call virtualViaPointer to print each Employee
32 // and earnings using dynamic binding
33 cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS\n";
34
35 for (const Employee* employeePtr : employees) {
36     virtualViaPointer(employeePtr);
37 }
38
39 // call virtualViaReference to print each Employee
40 // and earnings using dynamic binding
41 cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES\n";
42
43 for (const Employee* employeePtr : employees) {
44     virtualViaReference(*employeePtr); // note dereferenced pointer
45 }
46 }
47
48 // call Employee virtual functions toString and earnings via a
49 // base-class pointer using dynamic binding
50 void virtualViaPointer(const Employee* baseClassPtr) {
51     cout << fmt::format("{}\nearned ${:.2f}\n\n",
52                         baseClassPtr->toString(), baseClassPtr->earnings());
53 }
54
55 // call Employee virtual functions toString and earnings via a
56 // base-class reference using dynamic binding
57 void virtualViaReference(const Employee& baseClassRef) {
58     cout << fmt::format("{}\nearned ${:.2f}\n\n",
59                         baseClassRef.toString(), baseClassRef.earnings());
60 }
```

EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

```
1 // Fig. 10.19: Employee.h
2 // Employee abstract base class.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5 #include <string_view>
6
7 class Employee {
8 public:
9     Employee(std::string_view name);
10    virtual ~Employee() = default;
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    double earnings() const; // not virtual
16    std::string toString() const; // not virtual
17 protected:
18    virtual std::string getString() const; // virtual
19 private:
20    std::string m_name;
21    virtual double getPay() const = 0; // pure virtual
22};
```

```
1 // Fig. 10.20: Employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // constructor
9 Employee::Employee(string_view name) : m_name{name} {} // empty body
10
11 // set name
12 void Employee::setName(string_view name) {m_name = name;}
13
14 // get name
15 string Employee::getName() const {return m_name;}
16
17 // public non-virtual function; returns Employee's earnings
18 double Employee::earnings() const {return getPay();}
19
20 // public non-virtual function; returns Employee's string representation
21 string Employee::toString() const {return getString();}
22
23 // protected virtual function that derived classes can override and call
24 string Employee::getString() const {
25     return fmt::format("name: {}", getName());
26 }
```

```
1 // Fig. 10.21: SalariedEmployee.h
2 // SalariedEmployee class derived from Employee.
3 #pragma once
4 #include <string> // C++ standard string class
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class SalariedEmployee final : public Employee {
9 public:
10     SalariedEmployee(std::string_view name, double salary);
11     virtual ~SalariedEmployee() = default; // virtual destructor
12
13     void setSalary(double salary);
14     double getSalary() const;
15 private:
16     double m_salary{0.0};
17
18     // keyword override signals intent to override
19     double getPay() const override; // calculate earnings
20     std::string getString() const override; // string representation
21 };
```

```
1 // Fig. 10.23: CommissionEmployee.h
2 // CommissionEmployee class derived from Employee.
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Employee class definition
7
8 class CommissionEmployee final : public Employee {
9 public:
10     CommissionEmployee(std::string_view name, double grossSales,
11                         double commissionRate);
12     virtual ~CommissionEmployee() = default; // virtual destructor
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19 private:
20     double m_grossSales{0.0};
21     double m_commissionRate{0.0};
22
23     // keyword override signals intent to override
24     double getPay() const override; // calculate earnings
25     std::string getString() const override; // string representation
26 };
```

```
1 // Fig. 10.24: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "CommissionEmployee.h" // CommissionEmployee class definition
6 using namespace std;
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(string_view name,
10     double grossSales, double commissionRate) : Employee{name} {
11     setGrossSales(grossSales);
12     setCommissionRate(commissionRate);
13 }
14
15 // set gross sales amount
16 void CommissionEmployee::setGrossSales(double grossSales) {
17     if (grossSales < 0.0) {
18         throw invalid_argument("Gross sales must be >= 0.0");
19     }
20
21     m_grossSales = grossSales;
22 }
23
24 // return gross sales amount
25 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
26
27 // set commission rate
28 void CommissionEmployee::setCommissionRate(double commissionRate) {
29     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
30         throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
31     }
32
33     m_commissionRate = commissionRate;
34 }
35
36 // return commission rate
37 double CommissionEmployee::getCommissionRate() const {
38     return m_commissionRate;
39 }
40
41 // calculate earnings
42 double CommissionEmployee::getPay() const {
43     return getGrossSales() * getCommissionRate();
44 }
45
46 // return string representation of CommissionEmployee object
47 string CommissionEmployee::getString() const {
48     return fmt::format("{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::getString(),
49                         "gross sales", getGrossSales(),
50                         "commission rate", getCommissionRate());
51 }
```

EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES

name: John Smith
salary: \$800.00
earned \$800.00

name: Sue Jones
gross sales: \$10000.00
commission rate: 0.06
earned \$600.00

```
1 // Fig. 10.26: CompensationModel.h
2 // CompensationModel "interface" is a pure abstract base class.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5
6 class CompensationModel {
7 public:
8     virtual ~CompensationModel() = default; // generated destructor
9     virtual double earnings() const = 0; // pure virtual
10    virtual std::string toString() const = 0; // pure virtual
11};
```

```
1 // Fig. 10.27: Employee.h
2 // An Employee "has a" CompensationModel.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5 #include <string_view>
6 #include "CompensationModel.h"
7
8 class Employee final {
9 public:
10     Employee(std::string_view name, CompensationModel* modelPtr);
11     void setCompensationModel(CompensationModel *modelPtr);
12     double earnings() const;
13     std::string toString() const;
14 private:
15     std::string m_name{};
16     CompensationModel* m_modelPtr{}; // pointer to an implementation object
17 };
```

```
1 // Fig. 10.28: Employee.cpp
2 // Class Employee member-function definitions.
3 #include <string>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "CompensationModel.h"
6 #include "Employee.h"
7 using namespace std;
8
9 // constructor performs "constructor injection" to initialize
10 // the CompensationModel pointer to a CompensationModel implementation
11 Employee::Employee(string_view name, CompensationModel* modelPtr)
12     : m_name{name}, m_modelPtr{modelPtr} {}
13
14 // set function performs "property injection" to change the
15 // CompensationModel pointer to a new CompensationModel implementation
16 void Employee::setCompensationModel(CompensationModel* modelPtr) {
17     m_modelPtr = modelPtr;
18 }
19
20 // use the CompensationModel to calculate the Employee's earnings
21 double Employee::earnings() const {
22     return m_modelPtr->earnings();
23 }
24
25 // return string representation of Employee object
26 string Employee::toString() const {
27     return fmt::format("{}\n{}", m_name, m_modelPtr->toString());
28 }
```

```
1 // Fig. 10.29: Salaried.h
2 // Salaried implements the CompensationModel interface.
3 #pragma once
4 #include <string>
5 #include "CompensationModel.h" // CompensationModel definition
6
7 class Salaried final : public CompensationModel {
8 public:
9     explicit Salaried(double salary);
10    double earnings() const override;
11    std::string toString() const override;
12 private:
13     double m_salary{0.0};
14 };
```

```
1 // Fig. 10.30: Salaried.cpp
2 // Salaried compensation model member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Salaried.h" // class definition
6 using namespace std;
7
8 // constructor
9 Salaried::Salaried(double salary) : m_salary{salary} {
10     if (m_salary < 0.0) {
11         throw invalid_argument("Weekly salary must be >= 0.0");
12     }
13 }
14
15 // override CompensationModel pure virtual function earnings
16 double Salaried::earnings() const {return m_salary;}
17
18 // override CompensationModel pure virtual function toString
19 string Salaried::toString() const {
20     return fmt::format("salary: ${:.2f}", m_salary);
21 }
```

```
1 // Fig. 10.31: Commission.h
2 // Commission implements the CompensationModel interface.
3 #pragma once
4 #include <string>
5 #include "CompensationModel.h" // CompensationModel definition
6
7 class Commission final : public CompensationModel {
8 public:
9     Commission(double grossSales, double commissionRate);
10    double earnings() const override;
11    std::string toString() const override;
12 private:
13    double m_grossSales{0.0};
14    double m_commissionRate{0.0};
15};
```

```
1 // Fig. 10.32: Commission.cpp
2 // Commission member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Commission.h" // class definition
6 using namespace std;
7
8 // constructor
9 Commission::Commission(double grossSales, double commissionRate)
10    : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
11
12    if (m_grossSales < 0.0) {
13        throw invalid_argument("Gross sales must be >= 0.0");
14    }
15
16    if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
17        throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
18    }
19 }
20
21 // override CompensationModel pure virtual function earnings
22 double Commission::earnings() const {
23     return m_grossSales * m_commissionRate;
24 }
25
26 // override CompensationModel pure virtual function toString
27 string Commission::toString() const {
28     return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29                       m_grossSales, m_commissionRate);
30 }
```

```
1 // fig10_33.cpp
2 // Processing Employees with various CompensationModels.
3 #include <iostream>
4 #include <vector>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 #include "Employee.h"
7 #include "Salaried.h"
8 #include "Commission.h"
9 using namespace std;
10
11 int main() {
12     // create CompensationModels and Employees
13     Salaried salaried{800.0};
14     Employee salariedEmployee{"John Smith", &salaried};
15
16     Commission commission{10000, .06};
17     Employee commissionEmployee{"Sue Jones", &commission};
18
19     // create and initialize vector of Employees
20     vector employees{salariedEmployee, commissionEmployee};
21
22     // print each Employee's information and earnings
23     for (const Employee& employee : employees) {
24         cout << fmt::format("{}\nearned: ${:.2f}\n\n",
25                             employee.toString(), employee.earnings());
26     }
27 }
```

John Smith
salary: \$800.00
earned: \$800.00

Sue Jones
gross sales: \$10000.00; commission rate: 0.06
earned: \$600.00

```
1 // Fig. 10.34: Salaried.h
2 // Salaried compensation model.
3 #pragma once
4 #include <string>
5
6 class Salaried {
7 public:
8     Salaried(double salary);
9     double earnings() const;
10    std::string toString() const;
11 private:
12     double m_salary{0.0};
13 }
```

```
1 // Fig. 10.35: Salaried.cpp
2 // Salaried compensation model member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Salaried.h" // class definition
6 using namespace std;
7
8 // constructor
9 Salaried::Salaried(double salary) : m_salary{salary} {
10     if (m_salary < 0.0) {
11         throw invalid_argument("Weekly salary must be >= 0.0");
12     }
13 }
14
15 // calculate earnings
16 double Salaried::earnings() const {return m_salary;}
17
18 // return string containing Salaried compensation model information
19 string Salaried::toString() const {
20     return fmt::format("salary: ${:.2f}", m_salary);
21 }
```

```
1 // Fig. 10.36: Commission.h
2 // Commission compensation model.
3 #pragma once
4 #include <string>
5
6 class Commission {
7 public:
8     Commission(double grossSales, double commissionRate);
9     double earnings() const;
10    std::string toString() const;
11 private:
12    double m_grossSales{0.0};
13    double m_commissionRate{0.0};
14};
```

```
1 // Fig. 10.37: Commission.cpp
2 // Commission member-function definitions.
3 #include <stdexcept>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Commission.h" // class definition
6 using namespace std;
7
8 // constructor
9 Commission::Commission(double grossSales, double commissionRate)
10    : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
11
12    if (m_grossSales < 0.0) {
13        throw invalid_argument("Gross sales must be >= 0.0");
14    }
15
16    if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
17        throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
18    }
19}
20
21 // calculate earnings
22 double Commission::earnings() const {
23    return m_grossSales * m_commissionRate;
24}
25
26 // return string containing Commission information
27 string Commission::toString() const {
28    return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29                      m_grossSales, m_commissionRate);
30}
```

```
using CompensationModel = std::variant<Commission, Salaried>;
```

```
std::variant<Commission, Salaried>
```

```
1 // Fig. 10.38: Employee.h
2 // An Employee "has a" CompensationModel.
3 #pragma once // prevent multiple inclusions of header
4 #include <string>
5 #include <string_view>
6 #include <variant>
7 #include "Commission.h"
8 #include "Salaried.h"
9
10 // define a convenient name for the std::variant type
11 using CompensationModel = std::variant<Commission, Salaried>;
12
13 class Employee {
14 public:
15     Employee(std::string_view name, CompensationModel model);
16     void setCompensationModel(CompensationModel model);
17     double earnings() const;
18     std::string toString() const;
19 private:
20     std::string m_name{};
21     CompensationModel m_model; // note this is not a pointer
22 };
```

```
1 // Fig. 10.39: Employee.cpp
2 // Class Employee member-function definitions.
3 #include <string>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Employee.h"
6 using namespace std;
7
8 // constructor
9 Employee::Employee(string_view name, CompensationModel model)
10    : m_name{name}, m_model{model} {}
11
12 // change the Employee's CompensationModel
13 void Employee::setCompensationModel(CompensationModel model) {
14     m_model = model;
15 }
16
17 // return the Employee's earnings
18 double Employee::earnings() const {
19     auto getEarnings = [] (const auto& model){return model.earnings();};
20     return std::visit(getEarnings, m_model);
21 }
22
23 // return string representation of an Employee object
24 string Employee::toString() const {
25     auto getString = [] (const auto& model){return model.toString();};
26     return fmt::format("{}\n{}", m_name, std::visit(getString, m_model));
27 }
```

```
auto getEarnings = [](const auto& model){return model.earnings();};  
return std::visit(getEarnings, m_model);
```

```
1 // fig10_40.cpp
2 // Processing Employees with various compensation models.
3 #include <iostream>
4 #include <vector>
5 #include "fmt/format.h" // In C++20, this will be #include <format>
6 #include "Employee.h"
7 #include "Salaried.h"
8 #include "Commission.h"
9 using namespace std;
10
11 int main() {
12     Employee salariedEmployee{"John Smith", Salaried{800.0}};
13     Employee commissionEmployee{"Sue Jones", Commission{10000.0, .06}};
14
15     // create and initialize vector of three Employees
16     vector employees{salariedEmployee, commissionEmployee};
17
18     // print each Employee's information and earnings
19     for (const Employee& employee : employees) {
20         cout << fmt::format("{}\nearned: ${:.2f}\n\n",
21                             employee.toString(), employee.earnings());
22     }
23 }
```

```
John Smith
salary: $800.00
earned: $800.00
```

```
Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

```
1 // Fig. 10.41: Base1.h
2 // Definition of class Base1
3 #pragma once
4
5 // class Base1 definition
6 class Base1 {
7 public:
8     explicit Base1(int value) : m_value{value} {}
9     int getData() const {return m_value;}
10 private: // accessible to derived classes via getData member function
11     int m_value;
12 };
```

```
1 // Fig. 10.42: Base2.h
2 // Definition of class Base2
3 #pragma once
4
5 // class Base2 definition
6 class Base2 {
7 public:
8     explicit Base2(char letter) : m_letter{letter} {}
9     char getData() const {return m_letter;}
10 private: // accessible to derived classes via getData member function
11     char m_letter;
12 };
```

```
1 // Fig. 10.43: Derived.h
2 // Definition of class Derived which inherits
3 // multiple base classes (Base1 and Base2).
4 #pragma once
5
6 #include <iostream>
7 #include <string>
8 #include "Base1.h"
9 #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2 {
14 public:
15     Derived(int value, char letter, double real);
16     double getReal() const;
17     std::string toString() const;
18 private:
19     double m_real; // derived class's private data
20 };
```

```
1 // Fig. 10.44: Derived.cpp
2 // Member-function definitions for class Derived
3 #include "fmt/format.h" // In C++20, this will be #include <format>
4 #include "Derived.h"
5
6 // constructor for Derived calls Base1 and Base2 constructors
7 Derived::Derived(int value, char letter, double real)
8     : Base1{value}, Base2{letter}, m_real{real} {}
9
10 // return real
11 double Derived::getReal() const {return m_real;}
12
13 // display all data members of Derived
14 string Derived::toString() const {
15     return fmt::format("int: {}; char: {}; double: {}",
16                         Base1::getData(), Base2::getData(), getReal());
17 }
```

```
1 // fig10_45.cpp
2 // Driver for multiple-inheritance example.
3 #include <iostream>
4 #include "fmt/format.h" // In C++20, this will be #include <format>
5 #include "Base1.h"
6 #include "Base2.h"
7 #include "Derived.h"
8 using namespace std;
9
10 int main() {
11     Base1 base1{10}; // create Base1 object
12     Base2 base2{'Z'}; // create Base2 object
13     Derived derived{7, 'A', 3.5}; // create Derived object
14
15     // print data in each object
16     cout << fmt::format("{}: {}\n{}: {}\n{}: {}\n\n",
17                         "Object base1 contains", base1.getData(),
18                         "Object base2 contains the character", base2.getData(),
19                         "Object derived contains", derived.toString());
20
21     // print data members of derived-class object
22     // scope resolution operator resolves getData ambiguity
23     cout << fmt::format("{}\n{}: {}\n{}: {}\n{}: {}\n\n",
24                         "Data members of Derived can be accessed individually:",
25                         "int", derived.Base1::getData(),
26                         "char", derived.Base2::getData(),
27                         "double", derived.getReal());
```

```
28
29     cout << "Derived can be treated as an object of either base class:\n";
30
31 // treat Derived as a Base1 object
32 Base1* base1Ptr = &derived;
33 cout << fmt::format("base1Ptr->getData() yields {}\\n",
34                     base1Ptr->getData());
35
36 // treat Derived as a Base2 object
37 Base2* base2Ptr = &derived;
38 cout << fmt::format("base2Ptr->getData() yields {}\\n",
39                     base2Ptr->getData());
40 }
```

```
Object base1 contains: 10
Object base2 contains the character: Z
Object derived contains: int: 7; char: A; double: 3.5
```

```
Data members of Derived can be accessed individually:
int: 7
char: A
double: 3.5
```

```
Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

```
1 // fig10_46.cpp
2 // Attempting to polymorphically call a function that is
3 // inherited from each of two base classes.
4 #include <iostream>
5 using namespace std;
6
7 // class Base definition
8 class Base {
9 public:
10     virtual void print() const = 0; // pure virtual
11 };
12
13 // class DerivedOne definition
14 class DerivedOne : public Base {
15 public:
16     // override print function
17     void print() const override {cout << "DerivedOne\n";}
18 };
19
20 // class DerivedTwo definition
21 class DerivedTwo : public Base {
22 public:
23     // override print function
24     void print() const override {cout << "DerivedTwo\n";}
25 };
26
```

```
27 // class Multiple definition
28 class Multiple : public DerivedOne, public DerivedTwo {
29 public:
30     // qualify which version of function print
31     void print() const override {DerivedTwo::print();}
32 };
33
34 int main() {
35     Multiple both{}; // instantiate a Multiple object
36     DerivedOne one{}; // instantiate a DerivedOne object
37     DerivedTwo two{}; // instantiate a DerivedTwo object
38     Base* array[3]{}; // create array of base-class pointers
39
40     array[0] = &both; // ERROR--ambiguous
41     array[1] = &one;
42     array[2] = &two;
43
44     // polymorphically invoke print
45     for (int i{0}; i < 3; ++i) {
46         array[i] ->print();
47     }
48 }
```

Microsoft Visual C++ compiler error message:

```
c:\Users\PaulDeitel\Documents\examples\ch10\fig10_46\fig10_46.cpp(40,20):
error C2594: '=': ambiguous conversions from 'Multiple *' to 'Base *'
```

```
1 // fig10_47.cpp
2 // Using virtual base classes.
3 #include <iostream>
4 using namespace std;

5
6 // class Base definition
7 class Base {
8 public:
9     virtual void print() const = 0; // pure virtual
10 };
11
12 // class DerivedOne definition
13 class DerivedOne : virtual public Base {
14 public:
15     // override print function
16     void print() const override {cout << "DerivedOne\n";}
17 };
18
19 // class DerivedTwo definition
20 class DerivedTwo : virtual public Base {
21 public:
22     // override print function
23     void print() const override {cout << "DerivedTwo\n";}
24 };
25
```

```
26 // class Multiple definition
27 class Multiple : public DerivedOne, public DerivedTwo {
28 public:
29     // qualify which version of function print
30     void print() const override {DerivedTwo::print();}
31 };
32
33 int main() {
34     Multiple both; // instantiate Multiple object
35     DerivedOne one; // instantiate DerivedOne object
36     DerivedTwo two; // instantiate DerivedTwo object
37     Base* array[3];
38
39     array[0] = &both; // allowed now
40     array[1] = &one;
41     array[2] = &two;
42
43     // polymorphically invoke function print
44     for (int i = 0; i < 3; ++i) {
45         array[i]->print();
46     }
47 }
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

```
1 // fig11_01.cpp
2 // Standard library string class test program.
3 #include <iostream>
4 #include <string>
5 #include <string_view>
6 #include "fmt/format.h" // in C++20, this will be #include <format>
7 using namespace std;
8
9 int main() {
```

```
10 string s1{"happy"}; // initialize string from char*
11 string s2{" birthday"}; // initialize string from char*
12 string s3; // creates an empty string
13 string_view v{"hello"}; // initialize string_view from char*
14
15 // output strings and string_view
16 cout << fmt::format("s1: \"{}\"; s2: \"{}\"; s3: \"{}\"; v: \"{}\"\n\n",
17 s1, s2, s3, v);
18
```

```
s1: "happy"; s2: " birthday"; s3: ""; v: "hello"
```

```
19 // test overloaded equality and relational operators
20 cout << "The results of comparing s2 and s1:\n"
21     << fmt::format("s2 == s1: {}\\n", s2 == s1)
22     << fmt::format("s2 != s1: {}\\n", s2 != s1)
23     << fmt::format("s2 > s1: {}\\n", s2 > s1)
24     << fmt::format("s2 < s1: {}\\n", s2 < s1)
25     << fmt::format("s2 >= s1: {}\\n", s2 >= s1)
26     << fmt::format("s2 <= s1: {}\\n\\n", s2 <= s1);
27
```

The results of comparing s2 and s1:

```
s2 == s1: false
s2 != s1: true
s2 > s1: false
s2 < s1: true
s2 >= s1: false
s2 <= s1: true
```

```
28 // test string member function empty
29 cout << "Testing s3.empty():\n";
30
31 if (s3.empty()) {
32     cout << "s3 is empty; assigning s1 to s3;\n";
33     s3 = s1; // assign s1 to s3
34     cout << fmt::format("s3 is \"{}\"\n\n", s3);
35 }
36
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

```
37 // test overloaded string concatenation assignment operator
38 s1 += s2; // test overloaded concatenation
39 cout << fmt::format("s1 += s2 yields s1 = {}\n\n", s1);
40
41 // test string concatenation with a C string
42 s1 += " to you";
43 cout << fmt::format("s1 += \" to you\" yields s1 = {}\n\n", s1);
44
45 // test string concatenation with a C++14 string-object literal
46 s1 += ", have a great day!"s; // s after " for string-object literal
47 cout << fmt::format(
48     "s1 += \"", have a great day!"s yields\ns1 = {}\n\n", s1);
49
```

```
s1 += s2 yields s1 = happy birthday
```

```
s1 += " to you" yields s1 = happy birthday to you
```

```
s1 += ", have a great day!"s yields
s1 = happy birthday to you, have a great day!
```

```
50 // test string member function substr
51 cout << fmt::format("{} {}\n{}\\n\\n",
52                     "The substring of s1 starting at location 0 for",
53                     "14 characters, s1.substr(0, 14), is:", s1.substr(0, 14));
54
55 // test substr "to-end-of-string" option
56 cout << fmt::format("{} {}\n{}\\n\\n",
57                     "The substring of s1 starting at",
58                     "location 15, s1.substr(15), is:", s1.substr(15));
59
```

The substring of s1 starting at location 0 for 14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at location 15, s1.substr(15), is:
to you, have a great day!

```
60 // test copy constructor
61 string s4{s1};
62 cout << fmt::format("s4 = {}\n\n", s4);
63
```

s4 = happy birthday to you, have a great day!

```
64 // test overloaded copy assignment (=) operator with self-assignment
65 cout << "assigning s4 to s4\n";
66 s4 = s4;
67 cout << fmt::format("s4 = {}\n\n", s4);
68
```

```
assigning s4 to s4
s4 = happy birthday to you, have a great day!
```

```
69 // test string's string_view constructor
70 cout << "initializing s5 with string_view v\n";
71 string s5{v};
72 cout << fmt::format("s5 is {}\n\n", s5);
73
```

```
initializing s5 with string_view v
s5 is hello
```

```
74 // test using overloaded subscript operator to create lvalue
75 s1[0] = 'H';
76 s1[6] = 'B';
77 cout << fmt::format("{}:\n{}\n\n",
78                     "after s1[0] = 'H' and s1[6] = 'B', s1 is", s1);
79
```

```
after s1[0] = 'H' and s1[6] = 'B', s1 is:
Happy Birthday to you, have a great day!
```

```
80 // test index out of range with string member function "at"
81 try {
82     cout << "Attempt to assign 'd' to s1.at(100) yields:\n";
83     s1.at(100) = 'd'; // ERROR: subscript out of range
84 }
85 catch (const out_of_range& ex) {
86     cout << fmt::format("An exception occurred: {}\n", ex.what());
87 }
88 }
```

```
Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: basic_string::at: __n (which is 100) >= this->size()
(which is 40)
```

```
Time* timePtr{new Time{}};
```

```
Time* timePtr{new Time{12, 45, 0}};
```

```
int* gradesArray{new int[10]{}};
```

https://en.cppreference.com/w/cpp/language/value_initialization

```
1 // fig11_02.cpp
2 // Demonstrating unique_ptr.
3 #include <iostream>
4 #include <memory>
5 using namespace std;
6
7 class Integer {
8 public:
9     // constructor
10    Integer(int i) : value{i} {
11        cout << "Constructor for Integer " << value << "\n";
12    }
13
14     // destructor
15    ~Integer() {
16        cout << "Destructor for Integer " << value << "\n";
17    }
18
19    int getValue() const {return value;} // return Integer value
20 private:
21     int value{0};
22 };
23
24 // use unique_ptr to manipulate Integer object
25 int main() {
26     cout << "Creating a unique_ptr object that points to an Integer\n";
27
28     // create a unique_ptr object and "aim" it at a new Integer object
29     auto ptr{make_unique<Integer>(7)};
30
31     // use unique_ptr to call an Integer member function
32     cout << "Integer value: " << ptr->getValue()
33         << "\n\nMain ends\n";
34 }
```

Creating a unique_ptr object that points to an Integer
Constructor for Integer 7
Integer value: 7

Main ends
Destructor for Integer 7

```
auto ptr{make_unique<int[]>(10)};
```

```
1 // fig11_03.cpp
2 // MyArray class test program.
3 #include <iostream>
4 #include <stdexcept>
5 #include <utility> // for std::move
6 #include "MyArray.h"
7 using namespace std;
8
9 // function to return a MyArray by value
10 MyArray getArrayByValue() {
11     MyArray localInts{10, 20, 30}; // create three-element MyArray
12     return localInts; // return by value creates an rvalue
13 }
14
```

```
15 int main() {
16     MyArray ints1(7); // 7-element MyArray; note () rather than {}
17     MyArray ints2(10); // 10-element MyArray; note () rather than {}
18
19     // print ints1 size and contents
20     cout << "\nints1 size: " << ints1.size()
21         << "\ncontents: " << ints1; // uses overloaded <<
22
23     // print ints2 size and contents
24     cout << "\n\nints2 size: " << ints2.size()
25         << "\ncontents: " << ints2; // uses overloaded <<
26
```

```
MyArray(size_t) constructor
MyArray(size_t) constructor

ints1 size: 7
contents: {0, 0, 0, 0, 0, 0, 0}

ints2 size: 10
contents: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

```
27 // input and print ints1 and ints2
28 cout << "\n\nEnter 17 integers: ";
29 cin >> ints1 >> ints2; // uses overloaded >>
30
31 cout << "\nints1: " << ints1 << "ints2: " << ints2;
32
```

```
Enter 17 integers: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

```
ints1: {1, 2, 3, 4, 5, 6, 7}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

```
33 // use overloaded inequality (!=) operator
34 cout << "\n\nEvaluating: ints1 != ints2\n";
35
36 if (ints1 != ints2) {
37     cout << "ints1 and ints2 are not equal\n\n";
38 }
39
```

```
Evaluating: ints1 != ints2
ints1 and ints2 are not equal
```

```
40 // create MyArray ints3 by copying ints1
41 MyArray ints3{ints1}; // invokes copy constructor
42
43 // print ints3 size and contents
44 cout << "\nints3 size: " << ints3.size() << "\ncontents: " << ints3;
45
```

MyArray copy constructor

```
ints3 size: 7
contents: {1, 2, 3, 4, 5, 6, 7}
```

```
46 // use overloaded copy assignment (=) operator
47 cout << "\n\nAssigning ints2 to ints1:\n";
48 ints1 = ints2; // note target MyArray is smaller
49
50 cout << "\nints1: " << ints1 << "\nints2: " << ints2;
51
```

```
Assigning ints2 to ints1:
MyArray copy assignment operator
MyArray copy constructor
MyArray destructor
```

```
ints1: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

```
52 // use overloaded equality (==) operator
53 cout << "\n\nEvaluating: ints1 == ints2\n";
54
55 if (ints1 == ints2) {
56     cout << "ints1 and ints2 are equal\n\n";
57 }
58
```

```
Evaluating: ints1 == ints2
ints1 and ints2 are equal
```

```
59 // use overloaded subscript operator to create an rvalue
60 cout << "ints1[5] is " << ints1[5];
61
62 // use overloaded subscript operator to create an lvalue
63 cout << "\n\nAssigning 1000 to ints1[5]\n";
64 ints1[5] = 1000;
65 cout << "ints1: " << ints1;
66
67 // attempt to use out-of-range subscript
68 try {
69     cout << "\n\nAttempt to assign 1000 to ints1[15]\n";
70     ints1[15] = 1000; // ERROR: subscript out of range
71 }
72 catch (const out_of_range& ex) {
73     cout << "An exception occurred: " << ex.what() << "\n";
74 }
75
```

```
ints1[5] is 13
```

```
Assigning 1000 to ints1[5]
ints1: {8, 9, 10, 11, 12, 1000, 14, 15, 16, 17}
```

```
Attempt to assign 1000 to ints1[15]
An exception occurred: Index out of range
```

```
76 // initialize ints4 with contents of the MyArray returned by
77 // getArrayByValue; print size and contents
78 cout << "\nInitialize ints4 with temporary MyArray object\n";
79 MyArray ints4{getArrayByValue()};
80
81 cout << "\nints4 size: " << ints4.size() << "\ncontents: " << ints4;
82
```

Initialize ints4 with temporary MyArray object
MyArray(initializer_list) constructor

ints4 size: 3
contents: {10, 20, 30}

MyArray(initializer_list) constructor

```
83 // convert ints4 to an rvalue reference with std::move and
84 // use the result to initialize MyArray ints5
85 cout << "\n\nInitialize ints5 with the result of std::move(ints4)\n";
86 MyArray ints5{std::move(ints4)}; // invokes move constructor
87
88 cout << "\nints5 size: " << ints5.size() << "\ncontents: " << ints5;
89 cout << "\n\nSize of ints4 is now: " << ints4.size();
90
```

Initialize ints5 with the result of std::move(ints4)
MyArray move constructor

ints5 size: 3
contents: {10, 20, 30}

Size of ints4 is now: 0

```
91 // move contents of ints5 into ints4
92 cout << "\n\nMove ints5 into ints4 via move assignment\n";
93 ints4 = std::move(ints5); // invokes move assignment
94
95 cout << "\nints4 size: " << ints4.size() << "\ncontents: " << ints4;
96 cout << "\n\nSize of ints5 is now: " << ints5.size();
97
```

Move ints5 into ints4 via move assignment
MyArray move assignment operator

ints4 size: 3
contents: {10, 20, 30}

Size of ints5 is now: 0

```
98 // check if ints5 is empty by contextually converting it to a bool
99 if (ints5) {
100     cout << "\n\nints5 contains elements\n";
101 }
102 else {
103     cout << "\n\nints5 is empty\n";
104 }
105
```

```
ints5 is empty
```

```
numbers += 1 # Python does not have a ++ operator
```

```
106 // add one to every element of ints4 using preincrement  
107 cout << "\nints4: " << ints4;  
108 cout << "\npreincrementing ints4: " << ++ints4;  
109
```

```
ints4: {10, 20, 30}  
preincrementing ints4: {11, 21, 31}
```

```
110 // add one to every element of ints4 using postincrement
111 cout << "\n\npostincrementing ints4: " << ints4++ << "\n";
112 cout << "\nints4 now contains: " << ints4;
113
```

```
postincrementing ints4: MyArray copy constructor
{11, 21, 31}
MyArray destructor

ints4 now contains: {12, 22, 32}
```

```
114 // add a value to every element of ints4 using +=
115 cout << "\n\nAdd 7 to every ints4 element: " << (ints4 += 7) << "\n\n";
116 }
```

```
Add 7 to every ints4 element: {19, 29, 39}
```

```
1 // Fig. 11.4: MyArray.h
2 // MyArray class definition with overloaded operators.
3 #pragma once
4 #include <initializer_list>
5 #include <iostream>
6 #include <memory>
7
8 class MyArray final {
9     // overloaded stream extraction operator
10    friend std::istream& operator>>(std::istream& in, MyArray& a);
11
12    // used by copy assignment operator to implement copy-and-swap idiom
13    friend void swap(MyArray& a, MyArray& b) noexcept;
14
15 public:
16    explicit MyArray(size_t size); // construct a MyArray of size elements
17
18    // construct a MyArray with a braced-initializer list of ints
19    explicit MyArray(std::initializer_list<int> list);
20
21    MyArray(const MyArray& original); // copy constructor
22    MyArray& operator=(const MyArray& right); // copy assignment operator
23
24    MyArray(MyArray&& original) noexcept; // move constructor
25    MyArray& operator=(MyArray&& right) noexcept; // move assignment
26
27    ~MyArray(); // destructor
28
29    size_t size() const noexcept {return m_size;} // return size
30    std::string toString() const; // create string representation
31
```

```
32 // equality operator
33 bool operator==(const MyArray& right) const noexcept;
34
35 // subscript operator for non-const objects returns modifiable lvalue
36 int& operator[](size_t index);
37
38 // subscript operator for const objects returns non-modifiable lvalue
39 const int& operator[](size_t index) const;
40
41 // convert MyArray to a bool value: true if non-empty; false if empty
42 explicit operator bool() const noexcept {return size() != 0;}
43
44 // preincrement every element, then return updated MyArray
45 MyArray& operator++();
46
47 // postincrement every element, and return copy of original MyArray
48 MyArray operator++(int);
49
50 // add value to every element, then return updated MyArray
51 MyArray& operator+=(int value);
52 private:
53     size_t m_size{0}; // pointer-based array size
54     std::unique_ptr<int[]> m_ptr; // smart pointer to integer array
55 };
56
57 // overloaded operator<< is not a friend--does not access private data
58 std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

```
explicit MyArray(size_t size); // construct a MyArray of size elements
```

```
1 // Fig. 11.5: MyArray.cpp
2 // MyArray class member- and friend-function definitions.
3 #include <algorithm>
4 #include <initializer_list>
5 #include <iostream>
6 #include <memory>
7 #include <span>
8 #include <sstream>
9 #include <stdexcept>
10 #include <utility>
11 #include "fmt/format.h" // In C++20, this will be #include <format>
12 #include "MyArray.h" // MyArray class definition
13 using namespace std;
14
15 // MyArray constructor to create a MyArray of size elements containing 0
16 MyArray::MyArray(size_t size)
17     : m_size{size}, m_ptr{make_unique<int[]>(size)} {
18     cout << "MyArray(size_t) constructor\n";
19 }
20
```

```
array<int, 5> n{32, 27, 64, 18, 95};
```

```
explicit MyArray(std::initializer_list<int> list);
```

```
21 // MyArray constructor that accepts an initializer list
22 MyArray::MyArray(initializer_list<int> list)
23     : m_size{list.size()}, m_ptr{make_unique<int[]>(list.size())} {
24     cout << "MyArray(initializer_list) constructor\n";
25
26     // copy list argument's elements into m_ptr's underlying int array
27     // m_ptr.get() returns the int array's starting memory location
28     copy(begin(list), end(list), m_ptr.get());
29 }
30
```

```
MyArray(const MyArray& original); // copy constructor
```

```
31 // copy constructor: must receive a reference to a MyArray
32 MyArray::MyArray(const MyArray& original)
33     : m_size{original.size()}, m_ptr{make_unique<int[]>(original.size())} {
34     cout << "MyArray copy constructor\n";
35
36     // copy original's elements into m_ptr's underlying int array
37     const span<const int> source{original.m_ptr.get(), original.size()};
38     copy(begin(source), end(source), m_ptr.get());
39 }
40
```

```
MyArray& operator=(const MyArray& right); // copy assignment operator
```

```
41 // copy assignment operator: implemented with copy-and-swap idiom
42 MyArray& MyArray::operator=(const MyArray& right) {
43     cout << "MyArray copy assignment operator\n";
44     MyArray temp{right}; // invoke copy constructor
45     swap(*this, temp); // exchange contents of this object and temp
46     return *this;
47 }
48
```

```
MyArray(MyArray&& original) noexcept; // move constructor
MyArray& operator=(MyArray&& right) noexcept; // move assignment
```

```
49 // move constructor: must receive an rvalue reference to a MyArray
50 MyArray::MyArray(MyArray&& original) noexcept
51     : m_size{std::exchange(original.m_size, 0)},
52      m_ptr{std::move(original.m_ptr)} { // move original.m_ptr into m_ptr
53     cout << "MyArray move constructor\n";
54 }
55
```

```
56 // move assignment operator
57 MyArray& MyArray::operator=(MyArray&& right) noexcept {
58     cout << "MyArray move assignment operator\n";
59
60     if (this != &right) { // avoid self-assignment
61         // move right's data into this MyArray
62         m_size = std::exchange(right.m_size, 0); // indicate right is empty
63         m_ptr = std::move(right.m_ptr);
64     }
65
66     return *this; // enables x = y = z, for example
67 }
68
```

```
69 // destructor: This could be compiler-generated. We included it here so
70 // we could output when each MyArray is destroyed.
71 MyArray::~MyArray() {
72     cout << "MyArray destructor\n";
73 }
74
```

```
size_t size() const noexcept {return m_size;}; // return size
```

```
std::string toString() const; // create string representation
```

```
75 // return a string representation of a MyArray
76 string MyArray::toString() const {
77     const span<const int> items{m_ptr.get(), m_size};
78     ostringstream output;
79     output << "{";
80
81     // insert each item in the dynamic array into the ostringstream
82     for (size_t count{0}; const auto& item : items) {
83         ++count;
84         output << item << (count < m_size ? ", " : "");
85     }
86
87     output << "}";
88     return output.str();
89 }
90
```

```
bool operator==(const MyArray& right) const noexcept;
```

```
91 // determine if two MyArrays are equal and
92 // return true, otherwise return false
93 bool MyArray::operator==(const MyArray& right) const noexcept {
94     // compare corresponding elements of both MyArrays
95     const span<const int> lhs{m_ptr.get(), size()};
96     const span<const int> rhs{right.m_ptr.get(), right.size()};
97     return equal(begin(lhs), end(lhs), begin(rhs), end(rhs));
98 }
99
```

```
bool operator==(const MyArray& left, const MyArray& right) noexcept;
```

```
int& operator[](size_t index);
int operator[](size_t index) const;
```

```
100 // overloaded subscript operator for non-const MyArrays;
101 // reference return creates a modifiable lvalue
102 int& MyArray::operator[](size_t index) {
103     // check for index out-of-range error
104     if (index >= m_size) {
105         throw out_of_range("Index out of range");
106     }
107
108     return m_ptr[index]; // reference return
109 }
110
111 // overloaded subscript operator for const MyArrays
112 // const reference return creates a non-modifiable lvalue
113 const int& MyArray::operator[](size_t index) const {
114     // check for subscript out-of-range error
115     if (index >= m_size) {
116         throw out_of_range("Index out of range");
117     }
118
119     return m_ptr[index]; // returns copy of this element
120 }
121
```

```
explicit operator bool() const noexcept {return size() != 0;}
```

```
I22 // preincrement every element, then return updated MyArray
I23 MyArray& MyArray::operator++() {
I24     // use a span and for_each to increment every element
I25     const span<int> items{m_ptr.get(), m_size};
I26     for_each(begin(items), end(items), [](auto& item){++item;});
I27     return *this;
I28 }
I29
```

```
I30 // postincrement every element, and return copy of original MyArray
I31 MyArray MyArray::operator++(int) {
I32     MyArray temp(*this);
I33     ++(*this); // call preincrement operator++ to do the incrementing
I34     return temp; // return the temporary copy made before incrementing
I35 }
I36
```

```
MyArray& operator+=(int value);
```

```
I37 // add value to every element, then return updated MyArray
I38 MyArray& MyArray::operator+=(int value) {
I39     // use a span and for_each to increment every element
I40     const span<int> items{m_ptr.get(), m_size};
I41     for_each(begin(items), end(items),
I42         [value](auto& item) {item += value;});
I43     return *this;
I44 }
I45
```

```
friend std::istream& operator>>(std::istream& in, MyArray& a);
```

```
std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

```
146 // overloaded input operator for class MyArray;
147 // inputs values for entire MyArray
148 istream& operator>>(istream& in, MyArray& a) {
149     span<int> items{a.m_ptr.get(), a.m_size};
150
151     for (auto& item : items) {
152         in >> item;
153     }
154
155     return in; // enables cin >> x >> y;
156 }
157
```

```
158 // overloaded output operator for class MyArray
159 ostream& operator<<(ostream& out, const MyArray& a) {
160     out << a.toString();
161     return out; // enables cout << x << y;
162 }
163
```

```
friend HugeInt operator+(const HugeInt& left, int right);
friend HugeInt operator+(int left, const HugeInt& right);
```

```
friend void swap(MyArray& a, MyArray& b) noexcept;
```

```
164 // swap function used to implement copy-and-swap copy assignment operator
165 void swap(MyArray& a, MyArray& b) noexcept {
166     std::swap(a.m_size, b.m_size); // swap using std::swap
167     a.m_ptr.swap(b.m_ptr); // swap using unique_ptr swap member function
168 }
```

```
bool operator<=(const Time& right) const {
    return !(right < *this);
}
```

```
1 // fig11_06.cpp
2 // C++20 three-way comparison (spaceship) operator.
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include "fmt/format.h"
7 using namespace std;
8
9 class Time {
10 public:
11     Time(int hr, int min, int sec) noexcept
12         : m_hr{hr}, m_min{min}, m_sec{sec} {}
13
14     string toString() const {
15         return fmt::format("hr={}, min={}, sec={}", m_hr, m_min, m_sec);
16     }
17
18     // <=> operator automatically supports equality relational operators
19     auto operator<=>(const Time& t) const noexcept = default;
20 private:
21     int m_hr{0};
22     int m_min{0};
23     int m_sec{0};
24 };
25
26 int main() {
27     const Time t1(12, 15, 30);
28     const Time t2(12, 15, 30);
29     const Time t3(6, 30, 0);
30
31     cout << fmt::format("t1: {}\nt2: {}\nt3: {}\n\n",
32                         t1.toString(), t2.toString(), t3.toString());
33 }
```

```
t1: hr=12, min=15, sec=30
t2: hr=12, min=15, sec=30
t3: hr=6, min=30, sec=0
```

```
34 // using the equality and relational operators
35 cout << fmt::format("t1 == t2: {}\n", t1 == t2);
36 cout << fmt::format("t1 != t2: {}\n", t1 != t2);
37 cout << fmt::format("t1 < t2: {}\n", t1 < t2);
38 cout << fmt::format("t1 <= t2: {}\n", t1 <= t2);
39 cout << fmt::format("t1 > t2: {}\n", t1 > t2);
40 cout << fmt::format("t1 >= t2: {}\n\n", t1 >= t2);
41
42 cout << fmt::format("t1 == t3: {}\n", t1 == t3);
43 cout << fmt::format("t1 != t3: {}\n", t1 != t3);
44 cout << fmt::format("t1 < t3: {}\n", t1 < t3);
45 cout << fmt::format("t1 <= t3: {}\n", t1 <= t3);
46 cout << fmt::format("t1 > t3: {}\n", t1 > t3);
47 cout << fmt::format("t1 >= t3: {}\n\n", t1 >= t3);
48
```

```
t1 == t2: true
t1 != t2: false
t1 < t2: false
t1 <= t2: true
t1 > t2: false
t1 >= t2: true
```

```
t1 == t3: false
t1 != t3: true
t1 < t3: false
t1 <= t3: false
t1 > t3: true
t1 >= t3: true
```

```
49 // using <=> to perform comparisons
50 if ((t1 <=> t2) == 0) {
51     cout << "t1 is equal to t2\n";
52 }
53
54 if ((t1 <=> t3) > 0) {
55     cout << "t1 is greater than t3\n";
56 }
57
58 if ((t3 <=> t1) < 0) {
59     cout << "t3 is less than t1\n";
60 }
61 }
```

```
t1 is equal to t2
t1 is greater than t3
t3 is less than t1
```

```
if (ints1) { // if expects a condition
    ...
}
```

```
1 // fig11_07.cpp
2 // Single-argument constructors and implicit conversions.
3 #include <iostream>
4 #include "MyArray.h"
5 using namespace std;
6
7 void outputArray(const MyArray&); // prototype
8
9 int main() {
10     MyArray ints1(7); // 7-element MyArray
11     outputArray(ints1); // output MyArray ints1
12     outputArray(3); // convert 3 to a MyArray and output the contents
13 }
14
15 // print MyArray contents
16 void outputArray(const MyArray& arrayToOutput) {
17     cout << "The MyArray received has " << arrayToOutput.size()
18         << " elements. The contents are: " << arrayToOutput << "\n";
19 }
```

```
MyArray(size_t) constructor
The MyArray received has 7 elements. The contents are: {0, 0, 0, 0, 0, 0, 0}
MyArray(size_t) constructor
The MyArray received has 3 elements. The contents are: {0, 0, 0}
MyArray destructor
MyArray destructor
```

```
explicit MyArray(size_t size);
```

error: invalid initialization of reference of type ‘const MyArray&’
from expression of type ‘int’

```
1 // fig11_08.cpp
2 // Demonstrating an explicit constructor.
3 #include <iostream>
4 #include "MyArray.h"
5 using namespace std;
6
7 void outputArray(const MyArray&); // prototype
8
9 int main() {
10     MyArray ints1{7}; // 7-element MyArray
11     outputArray(ints1); // output MyArray ints1
12     outputArray(3); // convert 3 to a MyArray and output its contents
13     outputArray(MyArray(3)); // explicit single-argument constructor call
14 }
15
16 // print MyArray contents
17 void outputArray(const MyArray& arrayToOutput) {
18     cout << "The MyArray received has " << arrayToOutput.size()
19         << " elements. The contents are: " << arrayToOutput << "\n";
20 }
```

```
explicit operator bool() const noexcept;
```

```
static_cast<bool>(myArrayObject)
```

```
1 // Fig. 12.1: DivideByZeroException.h
2 // Class DivideByZeroException definition.
3 #include <stdexcept> // stdexcept header contains runtime_error
4
5 // DivideByZeroException objects should be thrown
6 // by functions upon detecting division-by-zero
7 class DivideByZeroException : public std::runtime_error {
8 public:
9     // constructor specifies default error message
10    DivideByZeroException()
11        : std::runtime_error{"attempted to divide by zero"} {}
12};
```

```
1 // fig12_02.cpp
2 // Example that throws an exception on
3 // an attempt to divide by zero.
4 #include <iostream>
5 #include "DivideByZeroException.h" // DivideByZeroException class
6 using namespace std;
7
8 // perform division and throw DivideByZeroException object if
9 // divide-by-zero exception occurs
10 double quotient(double numerator, double denominator) {
11     // throw DivideByZeroException if trying to divide by zero
12     if (denominator == 0.0) {
13         throw DivideByZeroException{}; // terminate function
14     }
15
16     // return division result
17     return numerator / denominator;
18 }
19
20 int main() {
21     int number1{0}; // user-specified numerator
22     int number2{0}; // user-specified denominator
23
24     cout << "Enter two integers (end-of-file to end): ";
25 }
```

```
26 // enable user to enter two integers to divide
27 while (cin >> number1 >> number2) {
28     // try block contains code that might throw exception
29     // and code that will not execute if an exception occurs
30     try {
31         double result{quotient(number1, number2)};
32         cout << "The quotient is: " << result << '\n';
33     }
34     catch (const DivideByZeroException& divideByZeroException) {
35         cout << "Exception occurred: "
36             << divideByZeroException.what() << '\n';
37     }
38
39     cout << "\nEnter two integers (end-of-file to end): ";
40 }
41
42 cout << '\n';
43 }
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857
```

```
Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero
```

```
Enter two integers (end-of-file to end): ^Z
```

```
throw out_of_range{"Index out of range"};
```

```
1 // fig12_03.cpp
2 // Rethrowing an exception.
3 #include <iostream>
4 #include <exception>
5 using namespace std;
6
7 // throw, catch and rethrow exception
8 void throwException() {
9     // throw exception and catch it immediately
10    try {
11        cout << " Function throwException throws an exception\n";
12        throw exception{}; // generate exception
13    }
14    catch (const exception&) { // handle exception
15        cout << " Exception handled in function throwException"
16            << "\n Function throwException rethrows exception";
17        throw; // rethrow exception for further processing
18    }
19
20    cout << "This should not print\n";
21 }
22
23 int main() {
24     // throw exception
25     try {
26         cout << "\nmain invokes function throwException\n";
27         throwException();
28         cout << "This should not print\n";
29     }
30     catch (const exception&) { // handle exception
31         cout << "\n\nException handled in main\n";
32     }
33
34     cout << "Program control continues after catch in main\n";
35 }
```

main invokes function throwException
Function throwException throws an exception
Exception handled in function throwException
Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main

```
1 // fig12_04.cpp
2 // Demonstrating stack unwinding.
3 #include <iostream>
4 #include <stdexcept>
5 using namespace std;
6
7 // function3 throws runtime error
8 void function3() {
9     cout << "In function 3\n";
10
11    // no try block, stack unwinding occurs, return control to function2
12    throw runtime_error{"runtime_error in function3"}; // no print
13 }
14
15 // function2 invokes function3
16 void function2() {
17     cout << "function3 is called inside function2\n";
18     function3(); // stack unwinding occurs, return control to function1
19 }
20
21 // function1 invokes function2
22 void function1() {
23     cout << "function2 is called inside function1\n";
24     function2(); // stack unwinding occurs, return control to main
25 }
26
27 // demonstrate stack unwinding
28 int main() {
29     // invoke function1
30     try {
31         cout << "function1 is called inside main\n";
32         function1(); // call function1 which throws runtime_error
33     }
34     catch (const runtime_error& error) { // handle runtime error
35         cout << "Exception occurred: " << error.what()
36             << "\nException handled in main\n";
37     }
38 }
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
terminate called after throwing an instance of 'std::runtime_error'
  what(): runtime_error in function3
Aborted (core dumped)
```

```
1 // fig12_05.cpp
2 // Demonstrating a function try block.
3 #include <iostream>
4 #include <limits>
5 #include <stdexcept>
6 using namespace std;
7
8 // class Integer purposely throws an exception from it's constructor
9 class Integer {
10 public:
11     explicit Integer(int i) : value{i} {
12         cout << "Integer constructor: " << value
13         << "\nPurposely throwing exception from Integer constructor\n";
14         throw runtime_error("Integer constructor failed");
15     }
16 private:
17     int value{};
18 };
19
20 class ResourceManager {
21 public:
22     ResourceManager(int i) try : myInteger(i) {
23         cout << "ResourceManager constructor called\n";
24     }
25     catch (runtime_error& ex) {
26         cout << "Exception while constructing ResourceManager: "
27             << ex.what() << "\nAutomatically rethrowing the exception\n";
28     }
29 private:
30     Integer myInteger;
31 };
32
33 int main() {
34     try {
35         const ResourceManager resource{7};
36     }
37     catch (const runtime_error& ex) {
38         cout << "Rethrown exception caught in main: " << ex.what() << "\n";
39     }
40 }
```

```
Integer constructor: 7
Purposely throwing exception from Integer constructor
Exception while constructing ResourceManager: Integer constructor failed
Automatically rethrowing the exception
Rethrown exception caught in main: Integer constructor failed
```

```
void myFunction() try {
    // do something
}
catch (const ExceptionType& ex) {
    // exception processing
}
```

```
void myFunction() {
    try {
        // do something
    }
    catch (const ExceptionType& ex) {
        // exception processing
    }
}
```

```
int* ptr{new int[100]}; // acquire dynamically allocated memory
processArray(ptr); // assume this function might throw an exception
delete[] ptr; // return the memory to the system
// ...
```

```
std::unique_ptr<int[]> ptr{std::make_unique<int>(100)};  
processArray(ptr); // possible exception here  
// ...
```

```
1 // fig12_06.cpp
2 // Demonstrating standard new throwing bad_alloc when memory
3 // cannot be allocated.
4 #include <array>
5 #include <iostream>
6 #include <memory>
7 #include <new> // bad_alloc class is defined here
8 using namespace std;
9
10 int main() {
11     array<unique_ptr<double[]>, 1000> items{};
12
13     // aim each unique_ptr at a big block of memory
14     try {
15         for (int i{0}; auto& item : items) {
16             item = make_unique<double[]>(500'000'000);
17             cout << "items[" << i++ << "] points to 500,000,000 doubles\n";
18         }
19     }
20     catch (const bad_alloc& memoryAllocationException) {
21         cerr << "Exception occurred: "
22             << memoryAllocationException.what() << '\n';
23     }
24 }
```

```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
items[8] points to 500,000,000 doubles
items[9] points to 500,000,000 doubles
Exception occurred: bad allocation
```

```
unique_ptr<double[]> ptr{new(nothrow) double[500'000'000]};
```

```
1 // fig12_07.cpp
2 // Demonstrating set_new_handler.
3 #include <array>
4 #include <iostream>
5 #include <memory>
6 #include <new> // set_new_handler is defined here
7 using namespace std;
8
9 // handle memory allocation failure
10 void customNewHandler() {
11     cerr << "customNewHandler was called\n";
12     exit(EXIT_FAILURE);
13 }
14
15 int main() {
16     array<unique_ptr<double[]>, 1000> items{};
17
18     // specify that customNewHandler should be called on
19     // memory allocation failure
20     set_new_handler(customNewHandler);
21
22     // aim each unique_ptr at a big block of memory
23     for (int i{0}; auto& item : items) {
24         item = make_unique<double[]>(500'000'000);
25         cout << "items[" << i++ << "] points to 500,000,000 doubles\n";
26     }
27 }
```

```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
items[8] points to 500,000,000 doubles
customNewHandler was called
```

<https://en.cppreference.com/w/cpp/error/exception>

```
double squareRoot(double value)
[[expects: value >= 0.0]];
```

```
[[assert: grade >= 0 && grade <= 100]];
```

<https://gitlab.com/lock3/gcc-new/-/wikis/contract-assertions>

```
[[pre default: denominator != 0.0]]
```

<https://godbolt.org/z/fwxE5ov9>

```
1 // fig12_08.cpp
2 // quotient function with a contract precondition.
3 #include <iostream>
4 using namespace std;
5
6 double quotient(double numerator, double denominator)
7     [[pre: denominator != 0.0]];
8
9 int main() {
10     cout << "quotient(100, 7): " << quotient(100, 7)
11     << "\nquotient(100, 0): " << quotient(100, 0) << '\n';
12 }
13
14 // perform division
15 double quotient(double numerator, double denominator) {
16     return numerator / denominator;
17 }
```

```
default std::handle_contractViolation called:  
./example.cpp 7 quotient denominator != 0.0 default default 0
```

```
-std=c++2a -fcontracts -fcontract-build-level=off
```

`-fcontract-continuation-mode=on`

<https://godbolt.org/z/K1qxf8MYY>

```
1 // fig12_09.cpp
2 // binarySearch function with a precondition and a postcondition.
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6 using namespace std;
7
8 template <typename T>
9 int binarySearch(const vector<T>& items, const T& key)
10    [[pre audit: is_sorted(begin(items), end(items))]]
11    [[post loc: loc == -1 || (loc >= 0 && loc < items.size())]];
12
13 int main() {
14     // sorted vector v1 satisfies binarySearch's precondition
15     vector v1{10, 20, 30, 40, 50, 60, 70, 80, 90};
16     int result1 = binarySearch(v1, 70);
17     cout << "70 was " << (result1 != -1 ? "" : "not ") << "found in v1\n";
18
19     // unsorted vector v2 violates binarySearch's precondition
20     vector v2{60, 70, 80, 90, 10, 20, 30, 40, 50};
21     int result2 = binarySearch(v2, 60);
22     cout << "60 was " << (result2 != -1 ? "" : "not ") << "found in v2\n";
23 }
```

```
[[pre audit: is_sorted(begin(items), end(items))]]
```

```
[[post loc: loc == -1 || (loc >= 0 && loc < items.size())]]
```

```
-std=c++2a -fcontracts -fcontract-build-level=audit
```

```
default std::handle_contractViolation called:  
./example.cpp 10 binarySearch<int> is_sorted(begin(items), end(items)) audit  
default 0
```

<https://godbolt.org/z/EbbYGET7n>

```
default std::handle_contractViolation called:  
./example.cpp 11 binarySearch<int> loc == -1 || (loc >= 0 &&  
loc < items.size()) default default 0
```

```
void handle_contractViolation(const contractViolation& violation)
{
    // handler code
}
```

<https://en.cppreference.com/w/cpp/algorithm>

https://en.cppreference.com/w/cpp/named_req

```
1 // fig13_01.cpp
2 // Demonstrating input and output with iterators.
3 #include <iostream>
4 #include <iterator> // ostream_iterator and istream_iterator
5
6 int main() {
7     std::cout << "Enter two integers: ";
8
9     // create istream_iterator for reading int values from cin
10    std::istream_iterator<int> inputInt{std::cin};
11
12    const int number1{*inputInt}; // read int from standard input
13    ++inputInt; // move iterator to next input value
14    const int number2{*inputInt}; // read int from standard input
15
16    // create ostream_iterator for writing int values to cout
17    std::ostream_iterator<int> outputInt{std::cout};
18
19    std::cout << "The sum is: ";
20    *outputInt = number1 + number2; // output result to cout
21    std::cout << "\n";
22 }
```

Enter two integers: 12 25

The sum is: 37

```
14 int main() {  
15     std::vector<int> integers{}; // create vector of ints  
16  
17     std::cout << "Size of integers: " << integers.size()  
18         << "\nCapacity of integers: " << integers.capacity() << "\n\n";  
19
```

```
Size of integers: 0  
Capacity of integers: 0
```

```
20 // append 1-10 to integers and display updated size and capacity
21 for (int i : std::views::iota(1, 11)) {
22     integers.push_back(i); // push_back is in vector, deque and list
23     showResult(i, integers.size(), integers.capacity());
24 }
25
```

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 3
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 6
appended: 6; size: 6; capacity: 6
appended: 7; size: 7; capacity: 9
appended: 8; size: 8; capacity: 9
appended: 9; size: 9; capacity: 9
appended: 10; size: 10; capacity: 13
```

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 4
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 8
appended: 6; size: 6; capacity: 8
appended: 7; size: 7; capacity: 8
appended: 8; size: 8; capacity: 8
appended: 9; size: 9; capacity: 16
appended: 10; size: 10; capacity: 16
```

```
26     std::cout << "\nOutput integers using iterators: ";
27
28     for (auto constIterator{integers.cbegin()});
29         constIterator != integers.cend(); ++constIterator) {
30         std::cout << *constIterator << ' ';
31     }
32 
```

```
Output integers using iterators: 1 2 3 4 5 6 7 8 9
```

```
for (auto const& item : integers) {  
    cout << item << ' ';  
}
```

```
33     std::cout << "\nOutput integers in reverse using iterators: ";
34
35 // display vector in reverse order using const_reverse_iterator
36 for (auto reverseIterator{integers.cbegin()});
37     reverseIterator != integers.crend(); ++reverseIterator) {
38     std::cout << *reverseIterator << ' ';
39 }
40
41 std::cout << "\n";
42 }
```

```
Output integers in reverse using iterators: 9 8 7 6 5 4 3 2 1
```

```
1 // fig13_03.cpp
2 // Testing standard library vector class template
3 // element-manipulation functions.
4 #include <algorithm> // copy algorithm
5 #include <fmt/format.h> // C++20: This will be #include <format>
6 #include <iostream>
7 #include <ranges>
8 #include <iterator> // ostream_iterator iterator
9 #include <vector>
10
11 int main() {
12     std::vector values{1, 2, 3, 4, 5}; // class template argument deduction
13     std::vector<int> integers{values.cbegin(), values.cend()};
14     std::ostream_iterator<int> output{std::cout, " "};
15 }
```

```
16     std::cout << "integers contains: ";
17     std::copy(integers.cbegin(), integers.cend(), output);
18
```

```
integers contains: 1 2 3 4 5
```

```
19     std::cout << fmt::format("\nfront: {}\nback: {}\n\n",
20                             integers.front(), integers.back());
21
```

```
front: 1
back: 5
```

```
22     integers[0] = 7; // set first element to 7
23     integers.at(2) = 10; // set element at position 2 to 10
24
```

```
25 // insert 22 as 2nd element
26 integers.insert(integers.cbegin() + 1, 22);
27
28 std::cout << "Contents of vector integers after changes: ";
29 std::ranges::copy(integers, output);
30
```

Contents of vector integers after changes: 7 22 2 10 4 5

```
31     integers.erase(integers.cbegin()); // erase first element
32     std::cout << "\n\nintegers after erasing first element: ";
33     std::ranges::copy(integers, output);
34
35     // erase remaining elements
36     integers.erase(integers.cbegin(), integers.cend());
37     std::cout << fmt::format("\nErased all elements: integers {} empty\n",
38                           integers.empty() ? "is" : "is not");
39
```

```
integers after erasing first element: 22 2 10 4 5
Erased all elements: integers is empty
```

```
40 // insert elements from the vector values
41 integers.insert(integers.cbegin(), values.cbegin(), values.cend());
42 std::cout << "\nContents of vector integers before clear: ";
43 std::ranges::copy(integers, output);
44
```

Contents of vector integers before clear: 1 2 3 4 5

```
45 // empty integers; clear calls erase to empty a collection
46 integers.clear();
47 std::cout << fmt::format("\nAfter clear, integers {} empty\n",
48                     integers.empty() ? "is" : "is not");
49 }
```

After clear, integers is empty

```
1 // fig13_04.cpp
2 // Standard library list class template.
3 #include <algorithm> // copy algorithm
4 #include <iostream>
5 #include <iterator> // ostream_iterator
6 #include <list> // list class-template definition
7 #include <vector>
8
9 // printList function template definition; uses
10 // ostream_iterator and copy algorithm to output list elements
11 template <typename T>
12 void printList(const std::list<T>& items) {
13     if (items.empty()) { // list is empty
14         std::cout << "List is empty";
15     }
16     else {
17         std::ostream_iterator<T> output{std::cout, " "};
18         std::ranges::copy(items, output);
19     }
20 }
21
```

```
22 int main() {
23     std::list<int> values{}; // create list of ints
24
25     // insert items in values
26     values.push_front(1);
27     values.push_front(2);
28     values.push_back(4);
29     values.push_back(3);
30
31     std::cout << "values contains: ";
32     printList(values);
33 }
```

```
values contains: 2 1 4 3
```

```
34     values.sort(); // sort values
35     std::cout << "\nvalues after sorting contains: ";
36     printList(values);
37
38
```

```
values after sorting contains: 1 2 3 4
```

```
39 // insert elements of ints into otherValues
40 std::vector<int> ints{2, 6, 4, 8};
41 std::list<int> otherValues{}; // create list of ints
42 otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
43 std::cout << "\nAfter insert, otherValues contains: ";
44 printList(otherValues);
45
46 // remove otherValues elements and insert at end of values
47 values.splice(values.cend(), otherValues);
48 std::cout << "\nAfter splice, values contains: ";
49 printList(values);
```

After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8

```
50     values.sort(); // sort values
51     std::cout << "\nAfter sort, values contains: ";
52     printList(values);
53
54     // insert elements of ints into otherValues
55     otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
56     otherValues.sort(); // sort the list
57     std::cout << "\nAfter insert and sort, otherValues contains: ";
58     printList(otherValues);
59
60     // remove otherValues elements and insert into values in sorted order
61     values.merge(otherValues);
62     std::cout << "\nAfter merge:\n  values contains: ";
63     printList(values);
64     std::cout << "\n  otherValues contains: ";
65     printList(otherValues);
66
```

After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
 values contains: 1 2 2 2 3 4 4 4 6 6 8 8
 otherValues contains: List is empty

```
67     values.pop_front(); // remove element from front
68     values.pop_back(); // remove element from back
69     std::cout << "\nAfter pop_front and pop_back:\n  values contains: ";
70     printList(values);
71
```

After pop_front and pop_back:
values contains: 2 2 2 3 4 4 4 6 6 8

```
72     values.unique(); // remove duplicate elements
73     std::cout << "\nAfter unique, values contains: ";
74     printList(values);
75 }
```

After unique, values contains: 2 3 4 6 8

```
76     values.swap(otherValues); // swap elements of values and otherValues
77     std::cout << "\nAfter swap:\n  values contains: ";
78     printList(values);
79     std::cout << "\n  otherValues contains: ";
80     printList(otherValues);
81
```

After swap:
values contains: List is empty
otherValues contains: 2 3 4 6 8

```
82 // replace contents of values with elements of otherValues
83 values.assign(otherValues.cbegin(), otherValues.cend());
84 std::cout << "\nAfter assign, values contains: ";
85 printList(values);
86
87 // remove otherValues elements and insert into values in sorted order
88 values.merge(otherValues);
89 std::cout << "\nAfter merge, values contains: ";
90 printList(values);
91
92 values.remove(4); // remove all 4s
93 std::cout << "\nAfter remove(4), values contains: ";
94 printList(values);
95 std::cout << "\n";
96 }
```

```
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8
```

```
1 // fig13_05.cpp
2 // Standard library deque class template.
3 #include <algorithm> // copy algorithm
4 #include <deque> // deque class-template definition
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7
8 int main() {
9     std::deque<double> values; // create deque of doubles
10    std::ostream_iterator<double> output{std::cout, " "};
11
12    // insert elements in values
13    values.push_front(2.2);
14    values.push_front(3.5);
15    values.push_back(1.1);
16
17    std::cout << "values contains: ";
18
19    // use subscript operator to obtain elements of values
20    for (size_t i{0}; i < values.size(); ++i) {
21        std::cout << values[i] << ' ';
22    }
23
24    values.pop_front(); // remove first element
25    std::cout << "\nAfter pop_front, values contains: ";
26    std::ranges::copy(values, output);
27
28    // use subscript operator to modify element at location 1
29    values[1] = 5.4;
30    std::cout << "\nAfter values[1] = 5.4, values contains: ";
31    std::ranges::copy(values, output);
32    std::cout << "\n";
33 }
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

```
std::multiset<int> ints{}; // multiset of int values
```

```
1 // fig13_06.cpp
2 // Standard library multiset class template
3 #include <algorithm> // copy algorithm
4 #include <fmt/format.h> // C++20: This will be #include <format>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <ranges>
8 #include <set> // multiset class-template definition
9 #include <vector>
10
11 int main() {
12     std::multiset<int, std::less<int>> ints{}; // multiset of int values
```

```
13     std::cout << fmt::format("15s in ints: {}\n", ints.count(15));  
14
```

15s in ints: 0

<https://en.cppreference.com/w/cpp/container/multiset/insert>

```
15     std::cout << "\nInserting two 15s into ints\n";
16     ints.insert(15); // insert 15 in ints
17     ints.insert(15); // insert 15 in ints
18     std::cout << fmt::format("15s in ints: {}\\n\\n", ints.count(15));
19
```

```
Inserting two 15s into ints
15s in ints: 2
```

```
20 // search for 15 and 20 in ints; find returns an iterator
21 for (int i : {15, 20}) {
22     if (auto result{ints.find(i)}; result != ints.end()) {
23         std::cout << fmt::format("Found {} in ints\n", i);
24     }
25     else {
26         std::cout << fmt::format("Did not find {} in ints\n", i);
27     }
28 }
29
```

```
Found 15 in ints
Did not find 20 in ints
```

```
30 // search for 15 and 20 in ints; contains returns a bool
31 for (int i : {15, 20}) {
32     if (ints.contains(i)) {
33         std::cout << fmt::format("Found {} in ints\n", i);
34     }
35     else {
36         std::cout << fmt::format("Did not find {} in ints\n", i);
37     }
38 }
39
```

```
Found 15 in ints
Did not find 20 in ints
```

```
std::ostream_iterator<int>{std::cout, " "}
```

```
40 // insert elements of vector values into ints
41 const std::vector values{7, 22, 9, 1, 18, 30, 100, 22, 85, 13};
42 ints.insert(values.cbegin(), values.cend());
43 std::cout << "\nAfter insert, ints contains:\n";
44 std::ranges::copy(ints, std::ostream_iterator<int>(std::cout, " "));
45
```

```
After insert, ints contains:
1 7 9 13 15 15 18 22 22 30 85 100
```

```
46 // determine lower and upper bound of 22 in ints
47 std::cout << fmt::format(
48     "\n\nlower_bound(22): {}\\nupper_bound(22): {}\\n\\n",
49     *(ints.lower_bound(22)), *(ints.upper_bound(22)));
50
```

```
lower_bound(22): 22
upper_bound(22): 30
```

```
51 // use equal_range to determine lower and upper bound of 22 in ints
52 auto p{ints.equal_range(22)};
53 std::cout << fmt::format(
54     "lower_bound(22): {}\nupper_bound(22): {}\n",
55     *(p.first), *(p.second));
56 }
```

```
lower_bound(22): 22
upper_bound(22): 30
```

```
std::set<double, std::less<double>> doubles{2.1, 4.2, 9.5, 2.1, 3.7};
```

```
1 // fig13_07.cpp
2 // Standard library set class template.
3 #include <algorithm>
4 #include <fmt/format.h> // C++20: This will be #include <format>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <set>
8
9 int main() {
10     std::set doubles{2.1, 4.2, 9.5, 2.1, 3.7}; // CTAD
11
12     std::ostream_iterator<double> output{std::cout, " "};
13     std::cout << "doubles contains: ";
14     std::ranges::copy(doubles, output);
15 }
```

```
doubles contains: 2.1 3.7 4.2 9.5
```

```
16 // insert 13.8 in doubles; insert returns pair in which
17 // p.first represents location of 13.8 in doubles and
18 // p.second represents whether 13.8 was inserted
19 auto p{doubles.insert(13.8)}; // value not in set
20 std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
21                         (p.second ? "was" : "was not"));
22 std::cout << "doubles contains: ";
23 std::ranges::copy(doubles, output);
24
```

```
13.8 was inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

```
25 // insert 9.5 in doubles
26 p = doubles.insert(9.5); // value already in set
27 std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
28                         (p.second ? "was" : "was not"));
29 std::cout << "doubles contains: ";
30 std::ranges::copy(doubles, output);
31 std::cout << "\n";
32 }
```

```
9.5 was not inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

```
std::multimap<int, double, std::less<int>> pairs{};
```

```
1 // fig13_08.cpp
2 // Standard library multimap class template.
3 #include <fmt/format.h> // C++20: This will be #include <format>
4 #include <iostream>
5 #include <map> // multimap class-template definition
6
7 int main() {
8     std::multimap<int, double, std::less<int>> pairs{}; // create multimap
```

```
9     std::cout << fmt::format("Number of 15 keys in pairs: {}\n",
10                  pairs.count(15));
11
```

There are currently 0 pairs with key 15 in the multimap

```
12 // insert two pairs
13 pairs.insert(std::make_pair(15, 99.3));
14 pairs.insert(std::make_pair(15, 2.7));
15 std::cout << fmt::format("Number of 15 keys in pairs: {}\\n\\n",
16                         pairs.count(15));
17
```

After inserts, there are 2 pairs with key 15

```
18 // insert five pairs
19 pairs.insert({30, 111.11});
20 pairs.insert({10, 22.22});
21 pairs.insert({25, 33.333});
22 pairs.insert({20, 9.345});
23 pairs.insert({5, 77.54});
24
25 std::cout << "Multimap pairs contains:\nKey\tValue\n";
26
27 // walk through elements of pairs
28 for (const auto& mapItem : pairs) {
29     std::cout << fmt::format("{}\t{}\n", mapItem.first, mapItem.second);
30 }
31 }
```

Multimap pairs contains:

Key	Value
5	77.54
10	22.22
15	99.3
15	2.7
20	9.345
25	33.333
30	111.11

```
std::multimap<int, double> pairs{  
    {10, 22.22}, {20, 9.345}, {5, 77.54}};
```

```
1 // fig13_09.cpp
2 // Standard library class map class template.
3 #include <iostream>
4 #include <fmt/format.h> // C++20: This will be #include <format>
5 #include <map> // map class-template definition
6
7 int main() {
8     // create a map; duplicate keys are ignored
9     std::map<int, double> pairs{{15, 2.7}, {30, 111.11}, {5, 1010.1},
10    {10, 22.22}, {25, 33.333}, {5, 77.54}, {20, 9.345}, {15, 99.3}};
11
12    // walk through elements of pairs
13    std::cout << "pairs contains:\nKey\tValue\n";
14    for (const auto& pair : pairs) {
15        std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
16    }
17
18    pairs[25] = 9999.99; // use subscripting to change value for key 25
19    pairs[40] = 8765.43; // use subscripting to insert value for key 40
20
21    // use const_iterator to walk through elements of pairs
22    std::cout << "\nAfter updates, pairs contains:\nKey\tValue\n";
23    for (const auto& pair : pairs) {
24        std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
25    }
26 }
```

```
pairs contains:
Key      Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11
```

```
After updates, pairs contains:
```

```
Key      Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

```
1 // fig13_10.cpp
2 // Standard library stack adaptor class.
3 #include <iostream>
4 #include <list> // list class-template definition
5 #include <ranges>
6 #include <stack> // stack adaptor definition
7 #include <vector> // vector class-template definition
8
9 // pushElements generic lambda to push values onto a stack
10 auto pushElements = [](auto& stack) {
11     for (auto i : std::views::iota(0, 10)) {
12         stack.push(i); // push element onto stack
13         std::cout << stack.top() << ' '; // view (and display) top element
14     }
15 };
16
17 // popElements generic lambda to pop elements off a stack
18 auto popElements = [](auto& stack) {
19     while (!stack.empty()) {
20         std::cout << stack.top() << ' '; // view (and display) top element
21         stack.pop(); // remove top element
22     }
23 };
24
25 int main() {
26     std::stack<int> dequeStack{}; // uses a deque by default
27     std::stack<int, std::vector<int>> vectorStack{}; // use a vector
28     std::stack<int, std::list<int>> listStack{}; // use a list
29 }
```

```
30 // push the values 0-9 onto each stack
31 std::cout << "Pushing onto dequeStack: ";
32 pushElements(dequeStack);
33 std::cout << "\nPushing onto vectorStack: ";
34 pushElements(vectorStack);
35 std::cout << "\nPushing onto listStack: ";
36 pushElements(listStack);
37
38 // display and remove elements from each stack
39 std::cout << "\n\nPopping from dequeStack: ";
40 popElements(dequeStack);
41 std::cout << "\nPopping from vectorStack: ";
42 popElements(vectorStack);
43 std::cout << "\nPopping from listStack: ";
44 popElements(listStack);
45 std::cout << "\n";
46 }
```

```
Pushing onto dequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto vectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto listStack: 0 1 2 3 4 5 6 7 8 9

Popping from dequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from vectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from listStack: 9 8 7 6 5 4 3 2 1 0
```

```
1 // fig13_11.cpp
2 // Standard library queue adaptor class template.
3 #include <iostream>
4 #include <queue> // queue adaptor definition
5
6 int main() {
7     std::queue<double> values{}; // queue with doubles
8
9     // push elements onto queue values
10    values.push(3.2);
11    values.push(9.8);
12    values.push(5.4);
13
14    std::cout << "Popping from values: ";
15
16    // pop elements from queue
17    while (!values.empty()) {
18        std::cout << values.front() << ' '; // view front element
19        values.pop(); // remove element
20    }
21
22    std::cout << "\n";
23 }
```

Popping from values: 3.2 9.8 5.4

```
1 // fig13_12.cpp
2 // Standard library priority_queue adaptor class.
3 #include <iostream>
4 #include <queue> // priority_queue adaptor definition
5
6 int main() {
7     std::priority_queue<double> priorities; // create priority_queue
8
9     // push elements onto priorities
10    priorities.push(3.2);
11    priorities.push(9.8);
12    priorities.push(5.4);
13
14    std::cout << "Popping from priorities: ";
15
16    // pop element from priority_queue
17    while (!priorities.empty()) {
18        std::cout << priorities.top() << ' '; // view top element
19        priorities.pop(); // remove top element
20    }
21
22    std::cout << "\n";
23 }
```

Popping from priorities: 9.8 5.4 3.2

<https://www.ssa.gov/employer/randomization.html>

<https://en.cppreference.com/w/cpp/algorithm>

<https://en.cppreference.com/w/cpp/iterator>

```
1 // fig14_01.cpp
2 // Lambda expressions.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{1, 2, 3, 4}; // initialize values
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14}
```

```
values contains: 1 2 3 4
```

```
std::copy(integers.cbegin(), integers.cend(), output);
```

```
15 // output each element multiplied by two
16 std::cout << "\nDisplay each element multiplied by two: ";
17 std::ranges::for_each(values, [](auto i) {std::cout << i * 2 << " "});
```

```
Display each element multiplied by two: 2 4 6 8
```

```
for_each(begin(items), end(items), [](auto& item){++item;});
```

<https://en.cppreference.com/w/cpp/iterator>

```
std::ranges::for_each(values, timesTwo);
```

```
19 // add each element to sum
20 int sum{0}; // initialize sum to zero
21 std::ranges::for_each(values, [&sum](auto i) {sum += i;});
22 std::cout << "\nSum of value's elements is: " << sum << "\n";
23 }
```

Sum of values

```
1 // fig14_02.cpp
2 // Algorithms fill, fill_n, generate and generate_n.
3 #include <algorithm> // algorithm definitions
4 #include <array> // array class-template definition
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7
8 // generator function returns next letter (starts with A)
9 char nextLetter() {
10     static char letter{'A'};
11     return letter++;
12 }
13
14 int main() {
15     std::array<char, 10> chars{};
```

```
16     std::ranges::fill(chars, '5'); // fill chars with 5s
17
18     std::cout << "chars after filling with 5s: ";
19     std::ostream_iterator<char> output{std::cout, " "};
20     std::ranges::copy(chars, output);
21
```

```
chars after filling with 5s: 5 5 5 5 5 5 5 5 5 5
```

```
22 // fill first five elements of chars with 'A's
23 std::ranges::fill_n(chars.begin(), 5, 'A');
24
25 std::cout << "\nchars after filling five elements with 'A's: ";
26 std::ranges::copy(chars, output);
27
```

```
chars after filling five elements with A A A A A 5 5 5 5 5
```

```
28 // generate values for all elements of chars with nextLetter
29 std::ranges::generate(chars, nextLetter);
30
31 std::cout << "\nchars after generating letters A-J: ";
32 std::ranges::copy(chars, output);
33
```

```
chars after generating letters A-J: A B C D E F G H I J
```

```
34 // generate values for first five elements of chars with nextLetter
35 std::ranges::generate_n(chars.begin(), 5, nextLetter);
36
37 std::cout << "\nchars after generating K-O into elements 0-4: ";
38 std::ranges::copy(chars, output);
39
```

```
chars after generating K-O into elements 0-4: K L M N O F G H I J
```

```
40 // generate values for first three elements of chars with a lambda
41 std::ranges::generate_n(chars.begin(), 3,
42     [](){ // lambda that takes no arguments
43         static char letter{'A'};
44         return letter++;
45     }
46 );
47
48 std::cout << "\nchars after generating A-C into elements 0-2: ";
49 std::ranges::copy(chars, output);
50 std::cout << "\n";
51 }
```

chars after generating A-C into elements 0-2: A B C N O F G H I J

```
1 // fig14_03.cpp
2 // Algorithms equal, mismatch and lexicographical_compare.
3 #include <algorithm> // algorithm definitions
4 #include <array> // array class-template definition
5 #include <fmt/format.h> // C++20: This will be #include <format>
6 #include <iomanip>
7 #include <iostream>
8 #include <iterator> // ostream_iterator
9 #include <string>
10
11 int main() {
12     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13     std::array a2{a1}; // initializes a2 with copy of a1
14     std::array a3{1, 2, 3, 4, 1000, 6, 7, 8, 9, 10};
15     std::ostream_iterator<int> output{std::cout, " "};
16
17     std::cout << "a1 contains: ";
18     std::ranges::copy(a1, output);
19     std::cout << "\na2 contains: ";
20     std::ranges::copy(a2, output);
21     std::cout << "\na3 contains: ";
22     std::ranges::copy(a3, output);
23 }
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 1 2 3 4 5 6 7 8 9 10
a3 contains: 1 2 3 4 1000 6 7 8 9 10
```

```
24 // compare a1 and a2 for equality
25 std::cout << fmt::format("\n\na1 is equal to a2: {}\n",
26                         std::ranges::equal(a1, a2));
27
28 // compare a1 and a3 for equality
29 std::cout << fmt::format("a1 is equal to a3: {}\n",
30                         std::ranges::equal(a1, a3));
31
```

```
a1 is equal to a2: true
a1 is equal to a3: false
```

```
32 // check for mismatch between a1 and a3
33 auto location{std::ranges::mismatch(a1, a3)};
34 std::cout << fmt::format("a1 and a3 mismatch at index {} ({} vs. {})\n",
35                         (location.in1 - a1.begin()),
36                         *location.in1, *location.in2);
37
```

```
a1 and a3 mismatch at index 4 (5 vs. 1000)
```

```
std::ranges::mismatch_result<borrowed_iterator_t<R1>,
borrowed_iterator_t<R2>>
```

```
std::ranges::mismatch_result<borrowed_iterator_t<array<int, 10>>,  
    borrowed_iterator_t<array<int, 10>>>
```

```
38     std::string s1{"HELLO"};
39     std::string s2{"BYE BYE"};
40
41     // perform lexicographical comparison of c1 and c2
42     std::cout << fmt::format("\\"{}\" < \"{}\": {}\\n", s1, s2,
43                           std::ranges::lexicographical_compare(s1, s2));
44 }
```

"HELLO" < "BYE BYE": false

```
1 // fig14_04.cpp
2 // Algorithms remove, remove_if, remove_copy and remove_copy_if.
3 #include <algorithm> // algorithm definitions
4 #include <iostream>
5 #include <iterator> // ostream_iterator
6 #include <vector>
7
8 int main() {
9     std::vector init{10, 2, 15, 4, 10, 6};
10    std::ostream_iterator<int> output{std::cout, " "};
11}
```

```
12 std::vector v1{init}; // initialize with copy of init
13 std::cout << "v1: ";
14 std::ranges::copy(v1, output);
15
16 // remove all 10s from v1
17 const auto& [begin1, end1]{std::ranges::remove(v1, 10)};
18 v1.erase(begin1, end1);
19 std::cout << "\nv1 after removing 10s: ";
20 std::ranges::copy(v1, output);
21
```

```
v1: 10 2 15 4 10 6
v1 after removing 10s: 2 15 4 6
```

```
v1.erase(std::remove(v1.begin(), v1.end(), 10), v1.end());
```

```
22     std::vector v2{init}; // initialize with copy of init
23     std::cout << "\n\nv2: ";
24     std::ranges::copy(v2, output);
25
26     // copy from v2 to c1, removing 10s in the process
27     std::vector<int> c1{};
28     std::ranges::remove_copy(v2, std::back_inserter(c1), 10);
29     std::cout << "\nc1 after copying v2 without 10s: ";
30     std::ranges::copy(c1, output);
31
```

```
v2: 10 2 15 4 10 6
c1 after copying v2 without 10s: 2 15 4 6
```

```
32     std::vector v3{init}; // initialize with copy of init
33     std::cout << "\n\nv3: ";
34     std::ranges::copy(v3, output);
35
36     // remove elements greater than 9 from a3
37     auto greaterThan9 = [] (auto x) {return x > 9;};
38     const auto& [first2, last2]{std::ranges::remove_if(v3, greaterThan9)};
39     v3.erase(first2, last2);
40     std::cout << "\nv3 after removing elements greater than 9: ";
41     std::ranges::copy(v3, output);
42
```

```
v3: 10 2 15 4 10 6
v3 after removing elements greater than 9: 2 4 6
```

```
43     std::vector v4{init}; // initialize with copy of init
44     std::cout << "\n\nv4: ";
45     std::ranges::copy(v4, output);
46
47     // copy elements from v4 to c2, removing elements greater than 9
48     std::vector<int> c2{}; // initialize to 0s
49     std::ranges::remove_copy_if(v4, std::back_inserter(c2), greaterThan9);
50     std::cout << "\nc2 after copying v4 without elements greater than 9: ";
51     std::ranges::copy(c2, output);
52     std::cout << "\n";
53 }
```

```
v4: 10 2 15 4 10 6
c2 after copying v4 without elements greater than 9: 2 4 6
```

```
1 // fig14_05.cpp
2 // Algorithms replace, replace_if, replace_copy and replace_copy_if.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7
8 int main() {
9     std::ostream_iterator<int> output{std::cout, " "};
10
```

```
11 std::array a1{10, 2, 15, 4, 10, 6};
12 std::cout << "a1: ";
13 std::ranges::copy(a1, output);
14
15 // replace all 10s in a1 with 100
16 std::ranges::replace(a1, 10, 100);
17 std::cout << "\na1 after replacing 10s with 100s: ";
18 std::ranges::copy(a1, output);
19
```

```
a1: 10 2 15 4 10 6
a1 after replacing 10s with 100s: 100 2 15 4 100 6
```

```
20     std::array a2{10, 2, 15, 4, 10, 6};
21     std::array<int, a2.size()> c1{};
22     std::cout << "\n\na2: ";
23     std::ranges::copy(a2, output);
24
25     // copy from a2 to c1, replacing 10s with 100s
26     std::ranges::replace_copy(a2, c1.begin(), 10, 100);
27     std::cout << "\nc1 after replacing a2's 10s with 100s: ";
28     std::ranges::copy(c1, output);
29
```

```
a2: 10 2 15 4 10 6
c1 after replacing a2
```

```
30     std::array a3{10, 2, 15, 4, 10, 6};
31     std::cout << "\n\na3: ";
32     std::ranges::copy(a3, output);
33
34     // replace values greater than 9 in a3 with 100
35     constexpr auto greaterThan9 = [] (auto x) {return x > 9;};
36     std::ranges::replace_if(a3, greaterThan9, 100);
37     std::cout << "\na3 after replacing values greater than 9 with 100s: ";
38     std::ranges::copy(a3, output);
39
```

```
a3: 10 2 15 4 10 6
```

```
a3 after replacing values greater than 9 with 100s: 100 2 100 4 100 6
```

```
40     std::array a4{10, 2, 15, 4, 10, 6};
41     std::array<int, a4.size()> c2{};
42     std::cout << "\n\na4: ";
43     std::ranges::copy(a4, output);
44
45     // copy a4 to c2, replacing elements greater than 9 with 100
46     std::ranges::replace_copy_if(a4, c2.begin(), greaterThan9, 100);
47     std::cout << "\nc2 after replacing a4's values "
48         << "greater than 9 with 100s: ";
49     std::ranges::copy(c2, output);
50     std::cout << "\n";
51 }
```

```
a4: 10 2 15 4 10 6
c2 after replacing a4
```

```
1 // fig14_06.cpp
2 // Mathematical algorithms of the standard library.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <numeric>
8 #include <random>
9
10 int main() {
11     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
12     std::ostream_iterator<int> output{std::cout, " "};
13
14     std::cout << "a1: ";
15     std::ranges::copy(a1, output);
16 }
```

```
a1: 1 2 3 4 5 6 7 8 9 10
```

```
default_random_engine randomEngine{random_device{}()};
```

```
17 // create random-number engine and use it to help shuffle a1
18 std::default_random_engine randomEngine{std::random_device{}()};
19 std::ranges::shuffle(a1, randomEngine); // randomly order elements
20 std::cout << "\na1 shuffled: ";
21 std::ranges::copy(a1, output);
22
```

```
a1 shuffled: 5 4 7 6 3 9 1 8 10 2
```

```
23     std::array a2{100, 2, 8, 1, 50, 3, 8, 8, 9, 10};
24     std::cout << "\n\na2: ";
25     std::ranges::copy(a2, output);
26
27     // count number of elements in a2 with value 8
28     auto result1{std::ranges::count(a2, 8)};
29     std::cout << "\nCount of 8s in a2: " << result1;
30
```

```
a2: 100 2 8 1 50 3 8 8 9 10
Count of 8s in a2: 3
```

```
31 // count number of elements in a2 that are greater than 9
32 auto result2{std::ranges::count_if(a2, [](auto x){return x > 9;})};
33 std::cout << "\nCount of a2 elements greater than 9: " << result2;
34
```

Count of a2 elements greater than 9: 3

```
35 // locate minimum element in a2
36 std::cout << "\n\na2 minimum element: "
37     << *(std::ranges::min_element(a2));
38
```

```
a2 minimum element: 1
```

```
39 // locate maximum element in a2
40 std::cout << "\na2 maximum element: "
41     << *(std::ranges::max_element(a2));
42
```

```
a2 maximum element: 100
```

```
43 // locate minimum and maximum elements in a2
44 const auto& [min, max]{std::ranges::minmax_element(a2)};
45 std::cout << "\na2 minimum and maximum elements: "
46     << *min << " and " << *max;
47
```

```
a2 minimum and maximum elements: 1 and 100
```

```
48 // calculate cube of each element in a1; place results in cubes
49 std::array<int, a1.size()> cubes{};
50 std::ranges::transform(a1, cubes.begin(),
51     [] (auto x){return x * x * x;});
52 std::cout << "\n\na1 values cubed: ";
53 std::ranges::copy(cubes, output);
54 std::cout << "\n";
55 }
```

```
a1 values cubed: 125 64 343 216 27 729 1 512 1000 8
```

```
1 // fig14_07.cpp
2 // Standard library search and sort algorithms.
3 #include <algorithm> // algorithm definitions
4 #include <array> // array class-template definition
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{10, 2, 17, 5, 16, 8, 13, 11, 20, 7};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output); // display output vector
14}
```

```
values contains: 10 2 17 5 16 8 13 11 20 7
```

```
15 // locate first occurrence of 16 in values
16 auto loc1{std::ranges::find(values, 16)};
17
18 if (loc1 != values.cend()) { // found 16
19     std::cout << "\n\nFound 16 at index: " << (loc1 - values.cbegin());
20 }
21 else { // 16 not found
22     std::cout << "\n\n16 not found";
23 }
24
25 // locate first occurrence of 100 in values
26 auto loc2{std::ranges::find(values, 100)};
27
28 if (loc2 != values.cend()) { // found 100
29     std::cout << "\n\nFound 100 at index: " << (loc2 - values.cbegin());
30 }
31 else { // 100 not found
32     std::cout << "\n\n100 not found";
33 }
34
```

Found 16 at index: 4
100 not found

```
35 // create variable to store lambda for reuse later
36 constexpr auto isGreaterThan10{[] (auto x){return x > 10;}};
37
38 // locate first occurrence of value greater than 10 in values
39 auto loc3{std::ranges::find_if(values, isGreaterThan10)};
40
41 if (loc3 != values.cend()) { // found value greater than 10
42     std::cout << "\n\nFirst value greater than 10: " << *loc3
43     << "\nfound at index: " << (loc3 - values.cbegin());
44 }
45 else { // value greater than 10 not found
46     std::cout << "\n\nNo values greater than 10 were found";
47 }
48
```

First value greater than 10: 17
found at index: 2

```
49 // sort elements of a
50 std::ranges::sort(values);
51 std::cout << "\n\nvalues after sort: ";
52 std::ranges::copy(values, output);
53
```

```
values after sort: 2 5 7 8 10 11 13 16 17 20
```

```
54 // use binary_search to check whether 13 exists in values
55 if (std::ranges::binary_search(values, 13)) {
56     std::cout << "\n\n13 was found in values";
57 }
58 else {
59     std::cout << "\n\n13 was not found in values";
60 }
61
62 // use binary_search to check whether 100 exists in values
63 if (std::ranges::binary_search(values, 100)) {
64     std::cout << "\n100 was found in values";
65 }
66 else {
67     std::cout << "\n100 was not found in values";
68 }
69
```

```
13 was found in values
100 was not found in values
```

```
70 // determine whether all of values' elements are greater than 10
71 if (std::ranges::all_of(values, isGreaterThan10)) {
72     std::cout << "\n\nAll values elements are greater than 10";
73 }
74 else {
75     std::cout << "\n\nSome values elements are not greater than 10";
76 }
77
```

Some values elements are not greater than 10

```
78 // determine whether any of values' elements are greater than 10
79 if (std::ranges::any_of(values, isGreaterThan10)) {
80     std::cout << "\n\nSome values elements are greater than 10";
81 }
82 else {
83     std::cout << "\n\nNo values elements are greater than 10";
84 }
85
```

Some values elements are greater than 10

```
86 // determine whether none of values' elements are greater than 10
87 if (std::ranges::none_of(values, isGreaterThan10)) {
88     std::cout << "\n\nNo values elements are greater than 10";
89 }
90 else {
91     std::cout << "\n\nSome values elements are greater than 10";
92 }
93
```

Some values elements are greater than 10

```
94 // locate first occurrence of value that
95 auto loc4{std::ranges::find_if_not(values, isGreaterThan10)};
96
97 if (loc4 != values.cend()) { // found a value less than or equal to 10
98     std::cout << "\n\nFirst value not greater than 10: " << *loc4
99     << "\nfound at index: " << (loc4 - values.cbegin());
100 }
101 else { // no values less than or equal to 10 were found
102     std::cout << "\n\nOnly values greater than 10 were found";
103 }
104
105 std::cout << "\n";
106 }
```

```
First value not greater than 10: 2
found at index: 0
```

```
1 // fig14_08.cpp
2 // Algorithms swap, iter_swap and swap_ranges.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9     std::array values{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14}
```

```
values contains: 1 2 3 4 5 6 7 8 9 10
```

```
15     std::swap(values[0], values[1]); // swap elements at index 0 and 1
16
17     std::cout << "\nvalues after swapping a[0] and a[1] with swap: ";
18     std::ranges::copy(values, output);
19
```

```
values after swapping a[0] and a[1] with swap: 2 1 3 4 5 6 7 8 9 10
```

```
20 // use iterators to swap elements at locations 0 and 1
21 std::iter_swap(values.begin(), values.begin() + 1);
22 std::cout << "\nvalues after swapping a[0] and a[1] with iter_swap: ";
23 std::ranges::copy(values, output);
24
```

```
values after swapping a[0] and a[1] with iter_swap: 1 2 3 4 5 6 7 8 9 10
```

```
25 // swap values and values2
26 std::array values2{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
27 std::cout << "\n\nBefore swap_ranges\nvalues contains: ";
28 std::ranges::copy(values, output);
29 std::cout << "\nvalues2 contains: ";
30 std::ranges::copy(values2, output);
31 std::ranges::swap_ranges(values, values2);
32 std::cout << "\n\nAfter swap_ranges\nvalues contains: ";
33 std::ranges::copy(values, output);
34 std::cout << "\nvalues2 contains: ";
35 std::ranges::copy(values2, output);
36
```

Before swap_ranges
values contains: 1 2 3 4 5 6 7 8 9 10
values2 contains: 10 9 8 7 6 5 4 3 2 1

After swap_ranges
values contains: 10 9 8 7 6 5 4 3 2 1
values2 contains: 1 2 3 4 5 6 7 8 9 10

```
37 // swap first five elements of values and values2
38 std::ranges::swap_ranges(values.begin(), values.begin() + 5,
39                         values2.begin(), values2.begin() + 5);
40
41 std::cout << "\n\nAfter swap_ranges for 5 elements"
42     << "\nvalues contains: ";
43 std::ranges::copy(values, output);
44 std::cout << "\nvalues2 contains: ";
45 std::ranges::copy(values2, output);
46 std::cout << "\n";
47 }
```

```
After swap_ranges for 5 elements
values contains: 1 2 3 4 5 5 4 3 2 1
values2 contains: 10 9 8 7 6 6 7 8 9 10
```

```
1 // fig14_09.cpp
2 // Algorithms copy_backward, merge, unique, reverse, copy_if and copy_n.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <vector>
8
9 int main() {
10     std::array a1{1, 3, 5, 7, 9};
11     std::array a2{2, 4, 5, 7, 9};
12     std::ostream_iterator<int> output{std::cout, " "};
13
14     std::cout << "array a1 contains: ";
15     std::ranges::copy(a1, output); // display a1
16     std::cout << "\narray a2 contains: ";
17     std::ranges::copy(a2, output); // display a2
18 }
```

```
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9
```

```
19 // place elements of a1 into results in reverse order
20 std::array<int, a1.size()> results{};
21 std::ranges::copy_backward(a1, results.end());
22 std::cout << "\n\nAfter copy_backward, results contains: ";
23 std::ranges::copy(results, output);
24
```

After copy_backward, results contains: 1 3 5 7 9

```
25 // merge elements of a1 and a2 into results2 in sorted order
26 std::array<int, a1.size() + a2.size()> results2{};
27 std::ranges::merge(a1, a2, results2.begin());
28
29 std::cout << "\n\nAfter merge of a1 and a2, results2 contains: ";
30 std::ranges::copy(results2, output);
31
```

After merge of a1 and a2, results2 contains: 1 2 3 4 5 5 7 7 9 9

```
32 // eliminate duplicate values from v
33 std::vector v(results2.begin(), results2.end());
34 const auto& [first, last]{std::ranges::unique(v)};
35 v.erase(first, last); // remove elements that no longer contain values
36
37 std::cout << "\n\nAfter unique v contains: ";
38 std::ranges::copy(v, output);
39
```

After unique, v contains: 1 2 3 4 5 7 9

```
40     std::cout << "\n\nAfter reverse, a1 contains: ";
41     std::ranges::reverse(a1); // reverse elements of a1
42     std::ranges::copy(a1, output);
43
```

After reverse, a1 contains: 9 7 5 3 1

```
44 // copy odd elements of a2 into v2
45 std::vector<int> v2{};
46 std::cout << "\n\nAfter copy_if, v2 contains: ";
47 std::ranges::copy_if(a2, std::back_inserter(v2),
48     [](auto x){return x % 2 == 0;});
49 std::ranges::copy(v2, output);
50
```

After copy_if, v2 contains: 2 4

```
51 // copy three elements of a2 into v3
52 std::vector<int> v3{};
53 std::cout << "\n\nAfter copy_n, v3 contains: ";
54 std::ranges::copy_n(a2.begin(), 3, std::back_inserter(v3));
55 std::ranges::copy(v3, output);
56 std::cout << "\n";
57 }
```

After copy_n, v3 contains: 2 4 5

```
1 // fig14_10.cpp
2 // Algorithms inplace_merge, reverse_copy and unique_copy.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <vector>
8
9 int main() {
10     std::array a1{1, 3, 5, 7, 9, 1, 3, 5, 7, 9};
11     std::ostream_iterator<int> output(std::cout, " ");
12
13     std::cout << "array a1 contains: ";
14     std::ranges::copy(a1, output);
15 }
```

```
array a1 contains: 1 3 5 7 9 1 3 5 7 9
```

```
16 // merge first half of a1 with second half of a1 such that
17 // a1 contains sorted set of elements after merge
18 std::ranges::inplace_merge(a1, a1.begin() + 5);
19 std::cout << "\nAfter inplace_merge, a1 contains: ";
20 std::ranges::copy(a1, output);
21
```

```
After inplace_merge, a1 contains: 1 1 3 3 5 5 7 7 9 9
```

```
22 // copy only unique elements of a1 into results1
23 std::vector<int> results1{};
24 std::ranges::unique_copy(a1, std::back_inserter(results1));
25 std::cout << "\nAfter unique_copy, results1 contains: ";
26 std::ranges::copy(results1, output);
27
```

```
After unique_copy results1 contains: 1 3 5 7 9
```

```
28 // copy elements of a1 into results2 in reverse order
29 std::vector<int> results2{};
30 std::ranges::reverse_copy(a1, std::back_inserter(results2));
31 std::cout << "\nAfter reverse_copy, results2 contains: ";
32 std::ranges::copy(results2, output);
33 std::cout << "\n";
34 }
```

```
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

```
1 // fig14_11.cpp
2 // Algorithms includes, set_difference, set_intersection,
3 // set_symmetric_difference and set_union.
4 #include <array>
5 #include <algorithm>
6 #include <fmt/format.h> // C++20: This will be #include <format>
7 #include <iostream>
8 #include <iterator>
9 #include <vector>
10
11 int main() {
12     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13     std::array a2{4, 5, 6, 7, 8};
14     std::array a3{4, 5, 6, 11, 15};
15     std::ostream_iterator<int> output{std::cout, " "};
16
17     std::cout << "a1 contains: ";
18     std::ranges::copy(a1, output); // display array a1
19     std::cout << "\na2 contains: ";
20     std::ranges::copy(a2, output); // display array a2
21     std::cout << "\na3 contains: ";
22     std::ranges::copy(a3, output); // display array a3
23 }
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
```

```
24 // determine whether a2 is completely contained in a1
25 std::cout << fmt::format("\n\na1 {}\n a2",
26                         std::ranges::includes(a1, a2) ?
27                             "includes" : "does not include");
28
29 // determine whether a3 is completely contained in a1
30 std::cout << fmt::format("\n\na1 {}\n a3",
31                         std::ranges::includes(a1, a3) ?
32                             "includes" : "does not include");
33
```

a1 includes a2

a1 does not include a3

```
34 // determine elements of a1 not in a2
35 std::vector<int> difference{};
36 std::ranges::set_difference(a1, a2, std::back_inserter(difference));
37 std::cout << "\n\nset_difference of a1 and a2 is: ";
38 std::ranges::copy(difference, output);
39
```

```
set_difference of a1 and a2 is: 1 2 3 9 10
```

```
40 // determine elements in both a1 and a2
41 std::vector<int> intersection{};
42 std::ranges::set_intersection(a1, a2,
43     std::back_inserter(intersection));
44 std::cout << "\n\nset_intersection of a1 and a2 is: ";
45 std::ranges::copy(intersection, output);
46
```

```
set_intersection of a1 and a2 is: 4 5 6 7 8
```

```
47 // determine elements of a1 that are not in a3 and
48 // elements of a3 that are not in a1
49 std::vector<int> symmetricDifference{};
50 std::ranges::set_symmetric_difference(a1, a3,
51     std::back_inserter(symmetricDifference));
52 std::cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
53 std::ranges::copy(symmetricDifference, output);
54
```

```
set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15
```

```
55 // determine elements that are in either or both sets
56 std::vector<int> unionSet{};
57 std::ranges::set_union(a1, a3, std::back_inserter(unionSet));
58 std::cout << "\n\nset_union of a1 and a3 is: ";
59 std::ranges::copy(unionSet, output);
60 std::cout << "\n";
61 }
```

```
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

```
1 // fig14_12.cpp
2 // Algorithms lower_bound, upper_bound and
3 // equal_range for a sorted sequence of values.
4 #include <algorithm>
5 #include <array>
6 #include <iostream>
7 #include <iterator>
8
9 int main() {
10     std::array values{2, 2, 4, 4, 4, 6, 6, 6, 6, 8};
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     std::cout << "values contains: ";
14     std::ranges::copy(values, output);
15 }
```

Array a contains: 2 2 4 4 4 6 6 6 6 8

```
16 // determine lower-bound insertion point for 6 in values
17 auto lower{std::ranges::lower_bound(values, 6)};
18 std::cout << "\n\nLower bound of 6 is index: "
19     << (lower - values.begin());
20
```

Lower bound of 6 is element 5 of array a

```
21 // determine upper-bound insertion point for 6 in values
22 auto upper{std::ranges::upper_bound(values, 6)};
23 std::cout << "\nUpper bound of 6 is index: "
24     << (upper - values.begin());
25
```

Upper bound of 6 is element 9 of array a

```
26 // use equal_range to determine the lower and upper bound of 6
27 const auto& [first, last]{std::ranges::equal_range(values, 6)};
28 std::cout << "\nUsing equal_range:\n    Lower bound of 6 is index: "
29     << (first - values.begin());
30 std::cout << "\n    Upper bound of 6 is index: "
31     << (last - values.begin());
32
```

Using equal_range:

Lower bound of 6 is element 5 of array a

Upper bound of 6 is element 9 of array a

```
33 // determine lower-bound insertion point for 3 in values
34 std::cout << "\n\nUse lower_bound to locate the first point "
35     << "at which 3 can be inserted in order";
36 lower = std::ranges::lower_bound(values, 3);
37 std::cout << "\n    Lower bound of 3 is index: "
38     << (lower - values.begin());
39
40 // determine upper-bound insertion point for 7 in values
41 std::cout << "\n\nUse upper_bound to locate the last point\n"
42     << "at which 7 can be inserted in order";
43 upper = std::ranges::upper_bound(values, 7);
44 std::cout << "\n    Upper bound of 7 is index: "
45     << (upper - values.begin());
46 }
```

Use lower_bound to locate the first point
at which 5 can be inserted in order
Lower bound of 5 is element 5 of array a

Use upper_bound to locate the last point
at which 7 can be inserted in order
Upper bound of 7 is element 9 of array a

```
1 // fig14_13.cpp
2 // Algorithms min, max, minmax and minmax_element.
3 #include <array>
4 #include <algorithm>
5 #include <iostream>
6
7 int main() {
8     std::cout << "Minimum of 12 and 7 is: " << std::min(12, 7)
9     << "\nMaximum of 12 and 7 is: " << std::max(12, 7)
10    << "\nMinimum of 'G' and 'Z' is: '" << std::min('G', 'Z') << ""
11    << "\nMaximum of 'G' and 'Z' is: '" << std::max('G', 'Z') << ""
12    << "\nMinimum of 'z' and 'Z' is: '" << std::min('z', 'Z') << "";
13}
```

```
Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12
Minimum of
Maximum of
Minimum of
```

```
14 // determine which argument is the min and which is the max
15 auto [smaller, larger]{std::minmax(12, 7)};
16     std::cout << "\n\nMinimum of 12 and 7 is: " << smaller
17         << "\nMaximum of 12 and 7 is: " << larger;
18
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
```

```
19     std::array items{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
20     std::ostream_iterator<int> output{std::cout, " "};
21
22     std::cout << "\n\nitems: ";
23     std::ranges::copy(items, output);
24
25     const auto& [smallest, largest]{std::ranges::minmax(items)};
26     std::cout << "\nMinimum value in items: " << smallest
27         << "\nMaximum value in items is: " << largest << "\n";
28 }
```

```
items: 3 100 52 77 22 31 1 98 13 40
Minimum value in items: 1
Maximum value in items is: 100
```

```
1 // fig14_14.cpp
2 // Demonstrating algorithms gcd, lcm, iota, reduce and partial_sum.
3 #include <array>
4 #include <algorithm>
5 #include <functional>
6 #include <iostream>
7 #include <iterator>
8 #include <numeric>
9
10 int main() {
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     // calculate the greatest common divisor of two integers
14     std::cout << "std::gcd(75, 20): " << std::gcd(75, 20)
15         << "\nstd::gcd(17, 13): " << std::gcd(75, 13);
16 }
```

```
std::gcd(75, 20): 5
std::gcd(17, 13): 1
```

```
17 // calculate the least common multiple of two integers
18 std::cout << "\n\nstd::lcm(3, 5): " << std::lcm(3, 5)
19     << "\nstd::lcm(12, 9): " << std::lcm(12, 9);
20
```

```
std::lcm(3, 5): 15
std::lcm(12, 9): 36
```

```
21 // fill an array with integers using the std::iota algorithm;
22 std::array<int, 5> ints{};
23 std::iota(ints.begin(), ints.end(), 1);
24 std::cout << "\n\nints: ";
25 std::ranges::copy(ints, output);
26
```

```
ints: 1 2 3 4 5
```

```
27 // reduce elements of a container to a single value
28 std::cout << "\n\nsum of ints: "
29     << std::reduce(ints.begin(), ints.end())
30     << "\nproduct of ints: "
31     << std::reduce(ints.begin(), ints.end(), 1, std::multiplies{});
```

32

```
sum of ints: 15
product of ints: 120
```

```
33 // calculate the partial sums of ints' elements
34 std::cout << "\n\nints: ";
35 std::ranges::copy(ints, output);
36 std::cout << "\n\npartial_sum of ints using std::plus by default: ";
37 std::partial_sum(ints.begin(), ints.end(), output);
38 std::cout << "\npartial_sum of ints using std::multiplies: ";
39 std::partial_sum(ints.begin(), ints.end(), output, std::multiplies{});
40 std::cout << "\n";
41 }
```

```
ints: 1 2 3 4 5
```

```
partial_sum of ints using std::plus by default: 1 3 6 10 15
```

```
partial_sum of ints using std::multiplies: 1 2 6 24 120
```

```
1 // fig14_15.cpp
2 // Algorithms push_heap, pop_heap, make_heap and sort_heap.
3 #include <iostream>
4 #include <algorithm>
5 #include <array>
6 #include <vector>
7 #include <iterator>
8
9 int main() {
10     std::ostream_iterator<int> output(std::cout, " ");
11 }
```

```
12     std::array heapArray{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
13     std::cout << "heapArray before make_heap:\n";
14     std::ranges::copy(heapArray, output);
15 
```

```
heapArray before make_heap:
3 100 52 77 22 31 1 98 13 40
```

```
16     std::ranges::make_heap(heapArray); // create heap from heapArray
17     std::cout << "\nheapArray after make_heap:\n";
18     std::ranges::copy(heapArray, output);
19
```

```
heapArray after make_heap:
100 98 52 77 40 31 1 3 13 22
```

```
20     std::ranges::sort_heap(heapArray); // sort elements with sort_heap
21     std::cout << "\nheapArray after sort_heap:\n";
22     std::ranges::copy(heapArray, output);
23
```

```
heapArray after sort_heap:
1 3 13 22 31 40 52 77 98 100
```

```
24 // lambda to add an int to a heap
25 auto push{
26     [&](std::vector<int>& heap, int value) {
27         std::cout << "\n\npushing " << value << " onto heap";
28         heap.push_back(value); // add value to the heap
29         std::ranges::push_heap(heap); // insert last element into heap
30         std::cout << "\nheap: ";
31         std::ranges::copy(heap, output);
32     }
33 };
34
```

```
35 // lambda to remove an item from the heap
36 auto pop{
37     [&](std::vector<int>& heap) {
38         std::ranges::pop_heap(heap); // remove max item from heap
39         std::cout << "\n\npopping highest priority item: " << heap.back();
40         heap.pop_back(); // remove vector's last element
41         std::cout << "\nheap: ";
42         std::ranges::copy(heap, output);
43     }
44 };
45
```

```
46     std::vector<int> heapVector{};  
47  
48     // place five integers into heapVector, maintaining it as a heap  
49     for (auto value : {3, 52, 100}) {  
50         push(heapVector, value);  
51     }  
52
```

```
pushing 3 onto heap  
heap: 3
```

```
pushing 52 onto heap  
heap: 52 3
```

```
pushing 100 onto heap  
heap: 100 3 52
```

```
53     pop(heapVector); // remove max item
54     push(heapVector, 22); // add new item to heap
55
```

```
popping highest priority item: 100
heap: 52 3
```

```
pushing 22 onto heap
heap: 52 3 22
```

```
56    pop(heapVector); // remove max item
57    push(heapVector, 77); // add new item to heap
58
```

```
popping highest priority item: 52
heap: 22 3
```

```
pushing 77 onto heap
heap: 77 3 22
```

```
59     pop(heapVector); // remove max item
60     pop(heapVector); // remove max item
61     pop(heapVector); // remove max item
62     std::cout << "\n";
63 }
```

```
popping highest priority item: 77
heap: 22 3
```

```
popping highest priority item: 22
heap: 3
```

```
popping highest priority item: 3
heap:
```

<https://en.cppreference.com/w/cpp/utility/functional>

```
1 // fig14_16.cpp
2 // Demonstrating function objects.
3 #include <array>
4 #include <algorithm>
5 #include <functional>
6 #include <iostream>
7 #include <iterator>
8 #include <numeric>
9
```

```
10 // binary function returns the sum of its first argument total
11 // and the square of its second argument value
12 int sumSquares(int total, int value) {
13     return total + value * value;
14 }
15
```

```
16 // class SumSquaresClass defines overloaded operator()
17 // that returns the sum of its first argument total
18 // and the square of its second argument value
19 class SumSquaresClass {
20 public:
21     // add square of value to total and return result
22     int operator()(int total, int value) {
23         return total + value * value;
24     }
25 };
26
```

```
27 int main() {
28     std::array integers{1, 2, 3, 4};
29     std::ostream_iterator<int> output{std::cout, " "};
30
31     std::cout << "array integers contains: ";
32     std::ranges::copy(integers, output);
33
34     // calculate sum of squares of elements of array integers
35     // using binary function sumSquares
36     int result{std::accumulate(integers.cbegin(), integers.cend(),
37         0, sumSquares)};
38
39     std::cout << "\n\nSum of squares\n"
40         << "via binary function sumSquares: " << result;
41
42     // calculate sum of squares of elements of array integers
43     // using binary function object
44     result = std::accumulate(integers.cbegin(), integers.cend(),
45         0, SumSquaresClass{});
46
47     std::cout << "\nvia a SumSquaresClass function object: " << result;
48
49     // calculate sum of squares array
50     result = std::accumulate(integers.cbegin(), integers.cend(),
51         0, [](auto total, auto value){return total + value * value;});
52
53     std::cout << "\nvia a lambda: " << result << "\n";
54 }
```

```
array integers contains: 1 2 3 4

Sum of squares
via binary function sumSquares: 30
via a SumSquaresClass function object: 30
via a lambda: 30
```

```
1 // fig14_17.cpp
2 // Demonstrating projections with C++20 range algorithms.
3 #include <array>
4 #include <algorithm>
5 #include <fmt/format.h>
6 #include <iostream>
7 #include <iterator>
8 #include <string>
9 #include <string_view>
10
11 class Employee {
12 public:
13     Employee(std::string_view first, std::string_view last, int salary)
14         : m_first{first}, m_last{last}, m_salary{salary} {}
15     std::string getLast() const {return m_last;}
16     std::string getFirst() const {return m_first;}
17     int getSalary() const {return m_salary;}
18 private:
19     std::string m_first;
20     std::string m_last;
21     int m_salary;
22 };
23
24 // operator<< for an Employee
25 std::ostream& operator<<(std::ostream& out, const Employee& e) {
26     out << fmt::format("{:10}{:10}{:10}", 
27                         e.getLast(), e.getFirst(), e.getSalary());
28     return out;
29 }
```

```
31 int main() {
32     std::array employees{
33         Employee{"Jason", "Red", 5000},
34         Employee{"Ashley", "Green", 7600},
35         Employee{"Matthew", "Indigo", 3587}
36     };
37
38     std::ostream_iterator<Employee> output{std::cout, "\n"};
39
40     std::cout << "Employees:\n";
41     std::ranges::copy(employees, output);
42 }
```

Employees:

Red	Jason	5000
Green	Ashley	7600
Indigo	Matthew	3587

```
43 // sort Employees by salary; {} indicates that the algorithm should
44 // use its default comparison function
45 std::ranges::sort(employees, {}),
46     [](const auto& e) {return e.getSalary();});
47 std::cout << "\nEmployees sorted in ascending order by salary:\n";
48 std::ranges::copy(employees, output);
49
```

```
Employees sorted in ascending order by salary:
Indigo    Matthew    3587
Red       Jason      5000
Green     Ashley     7600
```

```
std::ranges::sort(employees, {}, &Employee::getSalary);
```

```
50     // sort Employees by salary in descending order
51     std::ranges::sort(employees, std::ranges::greater{}, 
52                         &Employee::getSalary);
53     std::cout << "\nEmployees sorted in descending order by salary:\n";
54     std::ranges::copy(employees, output);
55 }
```

Employees sorted in descending order by salary:

Green	Ashley	7600
Red	Jason	5000
Indigo	Matthew	3587

```
1 // fig14_18.cpp
2 // Working with C++20 std::views.
3 #include <algorithm>
4 #include <iostream>
5 #include <iterator>
6 #include <map>
7 #include <ranges>
8 #include <string>
9 #include <vector>
10
11 int main() {
12     std::ostream_iterator<int> output{std::cout, " "};
13     auto isEven{[](int x) {return x % 2 == 0;}}; // true if x is even
14 }
```

```
15 // infinite view of even integers starting at 0
16 auto evens{std::views::iota(0) | std::views::filter(isEven)};
17
```

```
18     std::cout << "First five even ints: ";
19     std::ranges::copy(evens | std::views::take(5), output);
20
```

First five even ints: 0 2 4 6 8

```
21     std::cout << "\nEven ints less than 12: ";
22     auto lessThan12{
23         evens | std::views::take_while([](int x) {return x < 12;});
24     std::ranges::copy(lessThan12, output);
25 }
```

```
Even ints less than 12: 0 2 4 6 8 10
```

```
26     std::cout << "\nEven ints less than 12 reversed: ";
27     std::ranges::copy(lessThan12 | std::views::reverse, output);
28
```

```
Even ints less than 12 reversed: 10 8 6 4 2 0
```

```
29     std::cout << "\nSquares of even ints less than 12 reversed: ";
30     std::ranges::copy(
31         lessThan12
32             | std::views::reverse
33             | std::views::transform([](int x) {return x * x;}),
34         output);
35 
```

```
Squares of even ints less than 12 reversed: 100 64 36 16 4 0
```

```
36     std::cout << "\nSkip 1000 even ints, then take five: ";
37     std::ranges::copy(
38         evens | std::views::drop(1000) | std::views::take(5),
39         output);
40
```

```
Skip 1000 even ints, then take five: 2000 2002 2004 2006 2008
```

```
41     std::cout << "\nFirst five even ints greater than 1000: ";
42     std::ranges::copy(
43         evens
44             | std::views::drop_while([](int x) {return x <= 1000;})
45             | std::views::take(5),
46         output);
47
```

```
First five even ints greater than 1000: 1002 1004 1006 1008 1010
```

```
48 // allow std::string object literals
49 using namespace std::literals::string_literals;
50
51 std::map<std::string, int> romanNumerals{
52     {"I"s, 1}, {"II"s, 2}, {"III"s, 3}, {"IV"s, 4}, {"V"s, 5};
53     auto displayPair{}[](const auto& p) {
54         std::cout << p.first << " = " << p.second << "\n";
55     std::cout << "\n\nromanNumerals:\n";
56     std::ranges::for_each(romanNumerals, displayPair);
57 }
```

```
romanNumerals:
I = 1
II = 2
III = 3
IV = 4
V = 5
```

```
58     std::ostream_iterator<std::string> stringOutput{std::cout, " "};
59     std::cout << "\nKeys in romanNumerals: ";
60     std::ranges::copy(romanNumerals | std::views::keys, stringOutput);
61
62     std::cout << "\nValues in romanNumerals: ";
63     std::ranges::copy(romanNumerals | std::views::values, output);
64
```

Keys in romanNumerals: I II III IV V
Values in romanNumerals: 1 2 3 4 5

```
65     std::cout << "\nKeys in romanNumerals via std::views::elements: ";
66     std::ranges::copy(
67         romanNumerals | std::views::elements<0>, stringOutput);
68
69     std::cout << "\nvalues in romanNumerals via std::views::elements: ";
70     std::ranges::copy(romanNumerals | std::views::elements<1>, output);
71     std::cout << "\n";
72 }
```

```
Keys in romanNumerals via std::views::elements: I II III IV V
values in romanNumerals via std::views::elements: 1 2 3 4 5
```

<https://en.cppreference.com/w/cpp/algorithm>

<https://docs.microsoft.com/en-us/cpp/standard-library/algorithm>

<https://en.cppreference.com/w/cpp/algorithm/ranges>

<https://en.cppreference.com/w/cpp/experimental/parallelism>

<https://en.cppreference.com/w/cpp/header/memory>

```
1 // Fig. 15.1: Stack.h
2 // Stack class template.
3 #pragma once
4 #include <deque>
5
6 template<typename T>
7 class Stack {
8 public:
9     // return the top element of the Stack
10    const T& top() {
11        return stack.front();
12    }
13
14    // push an element onto the Stack
15    void push(const T& pushValue) {
16        stack.push_front(pushValue);
17    }
18
19    // pop an element from the stack
20    void pop() {
21        stack.pop_front();
22    }
23
24    // determine whether Stack is empty
25    bool isEmpty() const {
26        return stack.empty();
27    }
28
29    // return size of Stack
30    size_t size() const {
31        return stack.size();
32    }
33
34 private:
35     std::deque<T> stack{}; // internal representation of the Stack
36 };
```

```
1 // fig15_02.cpp
2 // Stack class template test program.
3 #include <iostream>
4 #include "Stack.h" // Stack class template definition
5 using namespace std;
6
7 int main() {
8     Stack<double> doubleStack{}; // create a Stack of double
9     constexpr size_t doubleStackSize{5}; // stack size
10    double doubleValue{1.1}; // first value to push
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    for (size_t i{0}; i < doubleStackSize; ++i) {
16        doubleStack.push(doubleValue);
17        cout << doubleValue << ' ';
18        doubleValue += 1.1;
19    }
20
21    cout << "\n\nPopping elements from doubleStack\n";
22
23    // pop elements from doubleStack
24    while (!doubleStack.isEmpty()) { // loop while Stack is not empty
25        cout << doubleStack.top() << ' '; // display top element
26        doubleStack.pop(); // remove top element
27    }
28}
```

```
29     cout << "\nStack is empty, cannot pop.\n";
30
31     Stack<int> intStack{}; // create a Stack of int
32     constexpr size_t intStackSize{10}; // stack size
33     int intValue{1}; // first value to push
34
35     cout << "\nPushing elements onto intStack\n";
36
37     // push 10 integers onto intStack
38     for (size_t i{0}; i < intStackSize; ++i) {
39         intStack.push(intValue);
40         cout << intValue++ << ' ';
41     }
42
43     cout << "\n\nPopping elements from intStack\n";
44
45     // pop elements from intStack
46     while (!intStack.isEmpty()) { // loop while Stack is not empty
47         cout << intStack.top() << ' '; // display top element
48         intStack.pop(); // remove top element
49     }
50
51     cout << "\nStack is empty, cannot pop.\n";
52 }
```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop.

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop.

```
template<typename T>
inline void Stack<T>::pop() {
    stack.pop_front();
}
```

```
template <typename T>
void printContainer(const T& items) {
    for (const auto& item : items) {
        std::cout << item << " ";
    }
}
```

```
1 // fig15_03.cpp
2 // Abbreviated function template.
3 #include <array>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 // abbreviated function template printContainer displays a
9 // container, such as an array or a vector
10 void printContainer(const auto& items) {
11     for (const auto& item : items) {
12         std::cout << item << " ";
13     }
14 }
15
16 int main() {
17     using namespace std::string_literals; // for string object literals
18
19     std::array ints{1, 2, 3, 4, 5};
20     std::vector strings{"red"s, "green"s, "blue"s};
21
22     std::cout << "ints: ";
23     printContainer(ints);
24     std::cout << "\nstrings: ";
25     printContainer(strings);
26     std::cout << "\n";
27 }
```

```
ints: 1 2 3 4 5
strings: red green blue
```

```
template <typename T>
T maximum(T value1, T value2, T value3) {
```

```
auto maximum(auto value1, auto value2, auto value3) {
```

```
[](auto total, auto value) {return total + value * value;}
```

```
[]<typename T>(T total, T value) {return total + value * value;}
```

```
1 // fig15_04.cpp
2 // Simple unconstrained multiply function template.
3 #include <iostream>
4
5 template<typename T>
6 T multiply(T first, T second) {
7     return first * second;
8 }
9
10 int main() {
11     std::cout << "Product of 5 and 3: " << multiply(5, 3)
12         << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
13 }
```

```
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
```

```
std::string s1{"hi"};
std::string s2{"bye"};
auto result{multiply(s1, s2)}; // error: string does not support *
```

```
fig15_04.cpp:7:17: error: invalid operands to binary expression (<char>
    return first * second;
           ~~~~ ^ ~~~~~

fig15_04.cpp:16:16: note: in instantiation of function template
specialization
    auto result{multiply(s1, s2)}; // error: string does not support *
           ^
1 error generated.
```

```
fig15_04.cpp
C:\Users\pauldeitel\examples\ch15\multiply\fig15_04.cpp(7,17): error C2676:
binary acceptable to the predefined operator
    with
    [
        T=std::string
    ]

C:\Users\pauldeitel\examples\ch15\multiply\fig15_04.cpp(16): message : see
reference to function template instantiation being compiled
    with
    [
        T=std::string
    ]
```

```
fig15_04.cpp: In instantiation of ‘T multiply(T, T) [with T =  
std::__cxx11::basic_string<char>]’:  
  
fig15_04.cpp:16:31:   required from here  
  
fig15_04.cpp:7:17: error: no match for ‘operator*’ (operand types are  
‘std::__cxx11::basic_string<char>’ and ‘std::__cxx11::basic_string<char>’)  
 7 |     return first * second;  
   |     ~~~~~~^~~~~~
```

```
std::list integers{10, 2, 33, 4, 7, 1, 80};  
std::sort(integers.begin(), integers.end());
```

```
1 // fig15_05.cpp
2 // Constrained multiply function template that allows
3 // only integers and floating-point values.
4 #include <concepts>
5 #include <iostream>
6
7 template<typename T>
8     requires std::integral<T> || std::floating_point<T>
9 T multiply(T first, T second) {
10     return first * second;
11 }
12
13 int main() {
14     std::cout << "Product of 5 and 3: " << multiply(5, 3)
15     << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
16
17     std::string s1{"hi"};
18     std::string s2{"bye"};
19     auto result{multiply(s1, s2)};
20 }
```

```
fig15_05.cpp: In function ‘int main()’:
fig15_05.cpp:19:31: error: use of function ‘T multiply(T, T) [with T =
std::__cxx11::basic_string<char>]’ with unsatisfied constraints
```

```
19 |     auto result{multiply(s1, s2)};           ^
|
```

```
fig15_05.cpp:9:3: note: declared here
9 | T multiply(T first, T second) {
| ^~~~~~
```

fig15_05.cpp:9:3: note: constraints not satisfied

```
fig15_05.cpp: In instantiation of ‘T multiply(T, T) [with T =
std::__cxx11::basic_string<char>]’:
fig15_05.cpp:19:31:   required from here
fig15_05.cpp:9:3:   required by the constraints of ‘template<class T>
requires (integral<T>) || (floating_point<T>) T multiply(T, T)’
```

fig15_05.cpp:8:30: note: no operand of the disjunction is satisfied

```
8 |     requires std::integral<T> || std::floating_point<T>
|           ~~~~~^~~~~~
```

```
1 // fig15_06.cpp
2 // Using type traits to test whether types are
3 // integral types, floating-point types or arithmetic types.
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <string>
7 #include <type_traits>
8
9 int main() {
10     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}\n",
11         "CHECK WITH TYPE TRAITS WHETHER TYPES ARE INTEGRAL",
12         "std::is_integral<int>::value: ", std::is_integral<int>::value,
13         "std::is_integral_v<int>: ", std::is_integral_v<int>,
14         "std::is_integral_v<long>: ", std::is_integral_v<long>,
15         "std::is_integral_v<float>: ", std::is_integral_v<float>,
16         "std::is_integral_v<std::string>: ",
17         std::is_integral_v<std::string>);
18
19     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}\n",
20         "CHECK WITH TYPE TRAITS WHETHER TYPES ARE FLOATING POINT",
21         "std::is_floating_point<float>::value: ",
22         std::is_floating_point<float>::value,
23         "std::is_floating_point_v<float>: ",
24         std::is_floating_point_v<float>,
25         "std::is_floating_point_v<double>: ",
26         std::is_floating_point_v<double>,
27         "std::is_floating_point_v<int>: ",
28         std::is_floating_point_v<int>,
29         "std::is_floating_point_v<std::string>: ",
30         std::is_floating_point_v<std::string>);
31 }
```

```
32     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n",
33         "CHECK WHETHER TYPES ARE INTEGRAL",
34         "std::is_integral<int>::value: ", std::is_integral<int>::value,
35         "std::is_integral_v<int>: ", std::is_integral_v<int>,
36         "std::is_integral_v<double>: ", std::is_integral_v<double>,
37         "std::is_integral_v<std::string>: ",
38         std::is_integral_v<std::string>);
39 }
```

CHECK WHETHER TYPES ARE INTEGRAL

```
std::is_integral<int>::value: true
std::is_integral_v<int>: true
std::is_integral_v<long>: true
std::is_integral_v<float>: false
std::is_integral_v<std::string>: false
```

CHECK WHETHER TYPES ARE FLOATING POINT

```
std::is_floating_point<float>::value: true
std::is_floating_point_v<float>: true
std::is_floating_point_v<double>: true
std::is_floating_point_v<int>: false
std::is_floating_point_v<std::string>: false
```

CHECK WHETHER TYPES CAN BE USED IN ARITHMETIC

```
std::is_arithmetic<int>::value: true
std::is_arithmetic_v<int>: true
std::is_arithmetic_v<double>: true
std::is_arithmetic_v<std::string>: false
```

```
std::is_integral<int>::value
```

```
std::is_integral<int>::value
```

```
template<class T>
inline constexpr bool is_integral_v = is_integral<T>::value;
```

https://en.cppreference.com/w/cpp/header/type_traits

```
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

```
template<typename T, typename U>
    requires std::same_as<T, U>
// rest of template definition
```

```
template<typename T>
    requires Numeric<T>
T multiply(T first, T second) {
    return first * second;
}
```

```
template<typename T>
T multiply(T first, T second) requires Numeric<T> {
    return first * second;
}
```

```
template<Numeric Number>
Number multiply(Number first, Number second) {
    return first * second;
}
```

```
1 // fig15_07.cpp
2 // Constrained multiply abbreviated function template.
3 #include <concepts>
4 #include <iostream>
5
6 // Numeric concept aggregates std::integral and std::floating_point
7 template<typename T>
8 concept Numeric = std::integral<T> || std::floating_point<T>;
9
10 // abbreviated function template with constrained auto
11 auto multiply(Numeric auto first, Numeric auto second) {
12     return first * second;
13 }
14
15 int main() {
16     std::cout << "Product of 5 and 3: " << multiply(5, 3)
17     << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0)
18     << "\nProduct of 5 and 7.25: " << multiply(5, 7.25) << "\n";
19 }
```

Product of 5 and 3: 15

Product of 7.25 and 2.0: 14.5

Product of 5 and 7.25: 36.25

```
template<Numeric Number1, Numeric Number2>
auto multiply(Number1 first, Number2 second) {
    return first * second;
}
```

```
1 // fig15_08.cpp
2 // Using concepts to select overloads.
3 #include <array>
4 #include <iostream>
5 #include <iterator>
6 #include <list>
7
8 // calculate the distance (number of items) between two iterators
9 // using input iterators; requires incrementing between iterators,
10 // so this is an O(n) operation
11 auto customDistance(std::input_iterator auto begin,
12                     std::input_iterator auto end) {
13     std::cout << "Called customDistance with bidirectional iterators\n";
14     std::ptrdiff_t count{0};
15
16     // increment from begin to end and count number of iterations
17     for (auto& iter{begin}; iter != end; ++iter) {
18         ++count;
19     }
20
21     return count;
22 }
23
24 // calculate the distance (number of items) between two iterators
25 // using random-access iterators and an O(1) operation
26 auto customDistance(std::random_access_iterator auto begin,
27                     std::random_access_iterator auto end) {
28     std::cout << "Called customDistance with random-access iterators\n";
29     return end - begin;
30 }
31
32 int main() {
33     std::array ints1{1, 2, 3, 4, 5}; // has random-access iterators
34     std::list ints2{1, 2, 3}; // has bidirectional iterators
35
36     auto result1{customDistance(ints1.begin(), ints1.end())};
37     std::cout << "ints1 number of elements: " << result1 << "\n";
38     auto result2{customDistance(ints2.begin(), ints2.end())};
39     std::cout << "ints2 number of elements: " << result2 << "\n";
40 }
```

```
Called customDistance with random-access iterators
ints1 number of elements: 5
Called customDistance with bidirectional iterators
ints2 number of elements: 3
```

```
1 template<class T>
2     concept range =
3         requires(T& t) {
4             std::ranges::begin(t);
5             std::ranges::end(t);
6         };

```

```
template<class T>
concept ArithmeticType =
    requires(T a, T b) {
        a + b;
        a - b;
        a * b;
        a / b;
        a += b;
        a -= b;
        a *= b;
        a /= b;
    };
```

```
template<typename T>
concept WorksWithVector = requires {
    typename std::vector<T>;
};
```

```
template<typename T>
requires requires(T& t) {
    std::ranges::begin(t);
    std::ranges::end(t);
}
void printRange(const T& range) {
    for (const auto& item : range) {
        std::cout << item << " ";
    }
}
```

```
template<class I>
concept has-arrow =
    input_iterator<I> && (is_pointer_v<I> ||  
    requires(I i) { i.operator->(); });
```

```
template<class I>
concept decrementable =
    incrementable<I> && requires(I i) {
        { --i } -> same_as<I&>;
        { i-- } -> same_as<I>;
    };
```

```
1 // fig15_09.cpp
2 // Testing custom concepts with static_assert.
3 #include <iostream>
4 #include <string>
5
6 template<typename T>
7 concept Numeric = std::integral<T> || std::floating_point<T>;
8
9 int main() {
10     static_assert(Numeric<int>); // OK: int is Numeric
11     static_assert(Numeric<double>); // OK: double is Numeric
12     static_assert(Numeric<std::string>); // error: string is not Numeric
13 }
```

```
fig15_09.cpp:12:4: error: static_assert failed
    static_assert(Numeric<std::string>); // error: string is not Numeric
    ^
~~~~~
```

```
fig15_09.cpp:12:18: note: because does not satisfy
    static_assert(Numeric<std::string>); // error: string is not Numeric
    ^
~~~~~
```

```
fig15_09.cpp:7:24: note: because
concept Numeric = std::integral<T> || std::floating_point<T>;
    ^
```

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../../include/c++/10/
concepts:102:24: note: because evaluated to false
    concept integral = is_integral_v<Tp>;
    ^
~~~~~
```

```
fig15_09.cpp:7:44: note: and
concept Numeric = std::integral<T> || std::floating_point<T>;
    ^
~~~~~
```

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../../include/c++/10/
concepts:111:30: note: because evaluated to false
    concept floating_point = is_floating_point_v<Tp>;
    ^
~~~~~
```

1 error generated.

```
1 // fig15_10.cpp
2 // A custom algorithm to calculate the average of
3 // a numeric input range's elements.
4 #include <algorithm>
5 #include <array>
6 #include <concepts>
7 #include <iostream>
8 #include <iterator>
9 #include <ranges>
10 #include <vector>
11
12 // concept for an input range containing integer or floating-point values
13 template<typename T>
14 concept NumericInputRange = std::ranges::input_range<T> &&
15     (std::integral<typename T::value_type> ||
16      std::floating_point<typename T::value_type>);
17
18 // calculate the average of a NumericInputRange's elements
19 auto average(NumericInputRange auto const& range) {
20     long double total{0};
21
22     for (auto i{range.begin()}; i != range.end(); ++i) {
23         total += *i; // dereference iterator and add value to total
24     }
25
26     // divide total by the number of elements in range
27     return total / std::ranges::distance(range);
28 }
29
```

```
30 int main() {
31     std::ostream_iterator<int> outputInt(std::cout, " ");
32     const std::array ints{1, 2, 3, 4, 5};
33     std::cout << "array ints: ";
34     std::ranges::copy(ints, outputInt);
35     std::cout << "\naverage of ints: " << average(ints);
36
37     std::ostream_iterator<double> outputDouble(std::cout, " ");
38     const std::vector doubles{10.1, 20.2, 35.3};
39     std::cout << "\n\nvector doubles: ";
40     std::ranges::copy(doubles, outputDouble);
41     std::cout << "\naverage of doubles: " << average(doubles);
42
43     std::ostream_iterator<long double> outputLongDouble(std::cout, " ");
44     const std::vector longDoubles{10.1L, 20.2L, 35.3L};
45     std::cout << "\n\nlist longDoubles: ";
46     std::ranges::copy(longDoubles, outputLongDouble);
47     std::cout << "\naverage of longDoubles: " << average(longDoubles)
48         << "\n";
49 }
```

```
array ints: 1 2 3 4 5
average of ints: 3

vector doubles: 10.1 20.2 35.3
average of doubles: 21.8667

list longDoubles: 10.1 20.2 35.3
average of doubles: 21.8667
```

<https://github.com/microsoft/STL/blob/main/stl/inc/array>

<https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array>

<https://github.com/llvm/llvm-project/blob/main/libcxx/include/array>

```
1 // Fig. 15.11: MyArray.h
2 // Class template MyArray with custom iterators implemented
3 // by class templates ConstIterator and Iterator
4 #pragma once
5 #include <algorithm>
6 #include <initializer_list>
7 #include <iostream>
8 #include <iterator>
9 #include <memory>
10 #include <stdexcept>
11 #include <utility>
12
13 // class template ConstIterator for a MyArray const iterator
14 template <typename T>
15 class ConstIterator {
16 public:
17     // public iterator nested type names
18     using iterator_category = std::bidirectional_iterator_tag;
19     using difference_type = std::ptrdiff_t;
20     using value_type = T;
21     using pointer = const value_type*;
22     using reference = const value_type&;
23
24     // default constructor
25     ConstIterator() : m_ptr{nullptr} {}
26
27     // initialize a ConstIterator with a pointer into a MyArray
28     ConstIterator(pointer p) : m_ptr{p} {}
29
```

```
30 // OPERATIONS ALL ITERATORS MUST PROVIDE
31 // increment the iterator to the next element and
32 // return a reference to the iterator
33 ConstIterator& operator++() noexcept {
34     ++m_ptr;
35     return *this;
36 }
37
38 // increment the iterator to the next element and
39 // return the iterator before the increment
40 ConstIterator operator++(int) noexcept {
41     ConstIterator temp{*this};
42     ++(*this);
43     return temp;
44 }
45
46 // OPERATIONS INPUT ITERATORS MUST PROVIDE
47 // return a const reference to the element m_ptr points to
48 reference operator*() const noexcept {return *m_ptr;}
49
50 // return a const pointer to the element m_ptr points to
51 pointer operator->() const noexcept {return m_ptr;}
52
53 // <=> operator automatically supports equality/relational operators.
54 // Only == and != are needed for bidirectional iterators.
55 // This implementation would support the <, <=, > and >= required
56 // by random-access iterators.
57 auto operator<=>(const ConstIterator& other) const noexcept = default;
58
59 // OPERATIONS BIDIRECTIONAL ITERATORS MUST PROVIDE
60 // decrement the iterator to the previous element and
61 // return a reference to the iterator
62 ConstIterator& operator--() noexcept {
63     --m_ptr;
64     return *this;
65 }
66
```

```
67     // decrement the iterator to the previous element and
68     // return the iterator before the deccrement
69     ConstIterator operator--(int) noexcept {
70         ConstIterator temp{*this};
71         --(*this);
72         return temp;
73     }
74 private:
75     pointer m_ptr;
76 };
77
```

```
78 // class template Iterator for a MyArray non-const iterator;
79 // redefines several inherited operators to return non-const results
80 template <typename T>
81 class Iterator : public ConstIterator<T> {
82 public:
83     // public iterator nested type names
84     using iterator_category = std::bidirectional_iterator_tag;
85     using difference_type = std::ptrdiff_t;
86     using value_type = T;
87     using pointer = value_type*;
88     using reference = value_type&;
89
90     // inherit ConstIterator constructors
91     using ConstIterator<T>::ConstIterator;
92
93     // OPERATIONS ALL ITERATORS MUST PROVIDE
94     // increment the iterator to the next element and
95     // return a reference to the iterator
96     Iterator& operator++() noexcept {
97         ConstIterator<T>::operator++(); // call base-class version
98         return *this;
99     }
100
101    // increment the iterator to the next element and
102    // return the iterator before the increment
103    Iterator operator++(int) noexcept {
104        Iterator temp{*this};
105        ConstIterator<T>::operator++(); // call base-class version
106        return temp;
107    }
108
```

```
109 // OPERATIONS INPUT ITERATORS MUST PROVIDE
110 // return a const reference to the element m_ptr points to; this
111 // operator returns a non-const reference for output iterator support
112 reference operator*() const noexcept {
113     return const_cast<reference>(ConstIterator<T>::operator*());
114 }
115
116 // return a const pointer to the element m_ptr points to
117 pointer operator->() const noexcept {
118     return const_cast<pointer>(ConstIterator<T>::operator->());
119 }
120
121 // OPERATIONS BIDIRECTIONAL ITERATORS MUST PROVIDE
122 // decrement the iterator to the previous element and
123 // return a reference to the iterator
124 Iterator& operator--() noexcept {
125     ConstIterator<T>::operator--(); // call base-class version
126     return *this;
127 }
128
129 // decrement the iterator to the previous element and
130 // return the iterator before the decrement
131 Iterator operator--(int) noexcept {
132     Iterator temp{*this};
133     ConstIterator<T>::operator--(); // call base-class version
134     return temp;
135 }
136 };
137
```

```
138 // class template MyArray contains a fixed-size T[SIZE] array;
139 // MyArray is an aggregate type with public data, like std::array
140 template <typename T, size_t SIZE>
141 struct MyArray {
142     // type names used in standard library containers
143     using value_type = T;
144     using size_type = size_t;
145     using difference_type = ptrdiff_t;
146     using pointer = value_type*;
147     using const_pointer = const value_type*;
148     using reference = value_type&;
149     using const_reference = const value_type&;
150
151     // iterator type names used in standard library containers
152     using iterator = Iterator<T>;
153     using const_iterator = ConstIterator<T>;
154     using reverse_iterator = std::reverse_iterator<iterator>;
155     using const_reverse_iterator = std::reverse_iterator<const_iterator>;
156
157     // Rule of Zero: MyArray
158
159     constexpr size_type size() const noexcept {return SIZE;} // return size
160
161     // member functions that return iterators
162     iterator begin() {return iterator{&m_data[0]};}
163     iterator end() {return iterator{&m_data[0] + size();}}
164     const_iterator begin() const {return const_iterator{&m_data[0]};}
165     const_iterator end() const {
166         return const_iterator{&m_data[0] + size();}
167     }
168     const_iterator cbegin() const {return begin();}
169     const_iterator cend() const {return end();}
170
```

```
171 // member functions that return reverse iterators
172 reverse_iterator rbegin() {return reverse_iterator{end();};}
173 reverse_iterator rend() {return reverse_iterator{begin();};}
174 const_reverse_iterator rbegin() const {
175     return const_reverse_iterator{cend();};
176 }
177 const_reverse_iterator rend() const {
178     return const_reverse_iterator{cbegin();};
179 }
180 const_reverse_iterator crbegin() const {return rbegin();}
181 const_reverse_iterator crend() const {return rend();}
182
183 // auto-generated three-way comparison operator
184 auto operator<=>(const MyArray& t) const noexcept = default;
185
186 // overloaded subscript operator for non-const MyArrays;
187 // reference return creates a modifiable lvalue
188 T& operator[](size_type index) {
189     // check for index out-of-range error
190     if (index >= SIZE) {
191         throw std::out_of_range("Index out of range");
192     }
193
194     return m_data[index]; // reference return
195 }
196
197 // overloaded subscript operator for const MyArrays;
198 // const reference return creates a non-modifiable lvalue
199 const T& operator[](size_type index) const {
200     // check for subscript out-of-range error
201     if (index >= SIZE) {
202         throw std::out_of_range("Index out of range");
203     }
204
205     return m_data[index]; // returns copy of this element
206 }
207
208 // like std::array the data is public to make this an aggregate type
209 T m_data[SIZE]; // built-in array of type T with SIZE elements
210 };
211
```

```
std::array ints{1, 2, 3, 4, 5};
```

```
212 // deduction guide to enable MyArrays to be brace initialized
213 template<typename T, typename... U>
214 MyArray(T first, U... rest) -> MyArray<T, 1 + sizeof...(U)>;
```

<https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array>

<https://github.com/llvm/llvm-project/blob/main/libcxx/include/array>

```
1 // fig15_12.cpp
2 // Using MyArray with range-based for and with
3 // C++ standard library algorithms.
4 #include <iostream>
5 #include <iterator>
6 #include "MyArray.h"
7
8 int main() {
9     std::ostream_iterator<int> outputInt{std::cout, " "};
10    std::ostream_iterator<double> outputDouble{std::cout, " "};
11    std::ostream_iterator<std::string> outputString{std::cout, " "};
12
13    std::cout << "Displaying MyArrays with std::ranges::copy, "
14        << "which requires input iterators:\n";
15    MyArray ints{1, 2, 3, 4, 5, 6, 7, 8};
16    std::cout << "ints: ";
17    std::ranges::copy(ints, outputInt);
18
19    MyArray doubles{1.1, 2.2, 3.3, 4.4, 5.5};
20    std::cout << "\ndoubles: ";
21    std::ranges::copy(doubles, outputDouble);
22
23    MyArray strings{"red", "orange", "yellow"};
24    std::cout << "\nstrings: ";
25    std::ranges::copy(strings, outputString);
26
27    std::cout << "\n\nDisplaying a MyArray with a range-based for "
28        << "statement, which requires input iterators:\n";
29    for (const auto& item : doubles) {
30        std::cout << item << " ";
31    }
32
```

```
33     std::cout << "\n\nCopying a MyArray with std::ranges::copy, "
34         << "which requires input iterators:\n";
35     MyArray<std::string, strings.size()> strings2{};
36     std::ranges::copy(strings, strings2.begin());
37     std::cout << "strings2 after copying from strings1: ";
38     std::ranges::copy(strings2, outputString);
39
40     std::cout << "\n\nFinding min and max elements in a MyArray "
41         << "with std::ranges::minmax_element, which requires "
42         << "forward iterators:\n";
43     const auto& [min, max] {std::ranges::minmax_element(strings)};
44     std::cout << "min and max elements of strings are: "
45         << *min << ", " << *max;
46
47     std::cout << "\n\nReversing a MyArray with std::ranges::reverse, "
48         << "which requires bidirectional iterators:\n";
49     std::ranges::reverse(ints);
50     std::cout << "ints after reversing elements: ";
51     std::ranges::copy(ints, outputInt);
52     std::cout << "\n";
53 }
```

Displaying MyArrays with std::ranges::copy, which requires input iterators:

ints: 1 2 3 4 5 6 7 8

doubles: 1.1 2.2 3.3 4.4 5.5

strings: red orange yellow

Displaying a MyArray with a range-based for statement, which requires input iterators:

1.1 2.2 3.3 4.4 5.5

Copying a MyArray with std::ranges::copy, which requires input iterators:

strings2 after copying from strings1: red orange yellow

Finding min and max elements in a MyArray with std::ranges::minmax_element, which requires forward iterators:

min and max elements of strings are: orange, yellow

Reversing a MyArray with std::ranges::reverse, which requires bidirectional iterators:

ints after reversing elements: 8 7 6 5 4 3 2 1

```
for (auto& item : std::as_const(doubles)) {
```

```
MyArray integers{10, 2, 33, 4, 7, 1, 80};  
std::ranges::sort(integers);
```

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2030:7: note: candidate template ignored: constraints not satisfied
[with _Range = MyArray<int, 7> &, _Comp = std::ranges::less, _Proj =
std::identity]
    operator()(_Range&& __r, _Comp __comp = {}, _Proj __proj = {}) const
    ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2026:14: note: because dom_access_range
template<random_access_range _Range,
          ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
range_access.h:924:37: note: because erator<int>
      = bidirectional_range<Tp> && random_access_iterator<iterator_t<Tp>>;
      ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/iter-
ator_concepts.h:591:10: note: because cept<Iterator<int> >, std::random_ac-
cess_iterator_tag>
      && derived_from<__detail::__iter_concept<_Iter>,
      ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:67:28: note: because std::bidirectional_iterator_tag)
    concept derived_from = __is_base_of(_Base, _Derived)
    ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
rang8least 2 arguments, but 1 was provided
    operator()(_Iter __first, _Sent __last,
    ^
1 error generated.
```

```
template <class T, class Container = deque<T>>
```

```
template<class T>
inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
```

```
1 // fig15_13.cpp
2 // Manipulating tuples.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6 #include <tuple>
7
8 // type alias for a tuple representing a hardware part's inventory
9 using Part = std::tuple<int, std::string, int, double>;
10
11 // return a part's inventory tuple
12 Part getInventory(int partNumber) {
13     using namespace std::string_literals; // for string object literals
14
15     switch (partNumber) {
16         case 1:
17             return {1, "Hammer"s, 32, 9.95}; // return a Part tuple
18         case 2:
19             return {2, "Screwdriver"s, 106, 6.99}; // return a Part tuple
20         default:
21             return {0, "INVALID PART"s, 0, 0.0}; // return a Part tuple
22     }
23 }
24 }
```

```
25 int main() {
26     // display the hardware part inventory
27     for (int i{1}; i <= 2; ++i) {
28         // unpack the returned tuple into four variables;
29         // variables' types are inferred from the tuple's element values
30         auto [partNumber, partName, quantity, price] {getInventory(i)};
31
32         std::cout << fmt::format("{}: {}, {}: {}, {}: {:.2f}\n",
33             "Part number", partNumber, "Tool", partName,
34             "Quantity", quantity, "Price", price);
35     }
36
37     std::cout << "\nAccessing a tuple's elements by index number:\n";
38     auto hammer{getInventory(1)};
39     std::cout << fmt::format("{}: {}, {}: {}, {}: {:.2f}\n",
40             "Part number", std::get<0>(hammer), "Tool", std::get<1>(hammer),
41             "Quantity", std::get<2>(hammer), "Price", std::get<3>(hammer));
42
43     std::cout << fmt::format("A Part tuple has {} elements\n",
44             std::tuple_size<Part>{}));
45 }
```

```
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
Part number: 2, Tool: Screwdriver, Quantity: 106, Price: 6.99
```

```
Accessing a tuple
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
A Part tuple has 4 elements
```

```
std::make_tuple(1, "Hammer"s, 32, 9.95)
```

```
auto partName{get<std::string>(hammerInventory)};
```

```
auto partNumber{get<int>(hammerInventory)};
```

<https://en.cppreference.com/w/cpp/utility/tuple/tuple>

<https://en.cppreference.com/w/cpp/utility/tuple>

```

1 // fig15_14.cpp
2 // Variadic function templates.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6
7 // base-case function for one argument
8 template <typename T>
9 auto sum(T item) {
10     return item;
11 }
12
13 // recursively add one or more arguments
14 template <typename FirstItem, typename... RemainingItems>
15 auto sum(FirstItem first, RemainingItems... theRest) {
16     return first + sum(theRest...); // expand parameter pack for next call
17 }
18
19 // add one or more arguments with a fold expression
20 template <typename FirstItem, typename... RemainingItems>
21 auto foldingSum(FirstItem first, RemainingItems... theRest) {
22     return (first + ... + theRest); // expand the parameter
23 }
24
25 int main() {
26     using namespace std::literals;
27
28     std::cout << "Recursive variadic function template sum:"
29         << fmt::format("\n{}{}\n{}{}\n{}{}\n{}\n",
30             "sum(1): ", sum(1), "sum(1, 2): ", sum(1, 2),
31             "sum(1, 2, 3): ", sum(1, 2, 3),
32             "sum(\"s\"s, \"u\"s, \"m\"s): ", sum("s"s, "u"s, "m"s));
33
34     std::cout << "Variadic function template foldingSum:"
35         << fmt::format("\n{}{}\n{}{}\n{}{}\n{}\n",
36             "sum(1): ", foldingSum(1), "sum(1, 2): ", foldingSum(1, 2),
37             "sum(1, 2, 3): ", foldingSum(1, 2, 3),
38             "sum(\"s\"s, \"u\"s, \"m\"s): ",
39             foldingSum("s"s, "u"s, "m"s));
40 }

```

Recursive variadic function template sum:

```

sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

```

Variadic function template foldingSum:

```

sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

```

```
sum("s"s, sum("u"s, sum("m"s)))
```

```
1 // fig15_15.cpp
2 // Unary fold expressions.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 template <typename... Items>
7 auto unaryLeftAdd(Items... items) {
8     return (... + items); // unary left fold
9 }
10
11 template <typename... Items>
12 auto unaryRightAdd(Items... items) {
13     return (items + ...); // unary right fold
14 }
15
16 template <typename... Items>
17 auto unaryLeftSubtract(Items... items) {
18     return (... - items); // unary left fold
19 }
20
21 template <typename... Items>
22 auto unaryRightSubtract(Items... items) {
23     return (items - ...); // unary right fold
24 }
25
26 int main() {
27     std::cout << "Unary left and right fold with addition:"
28     << fmt::format("\n{}{}\n{}{}\n\n",
29                     "unaryLeftAdd(1, 2, 3, 4): ", unaryLeftAdd(1, 2, 3, 4),
30                     "unaryRightAdd(1, 2, 3, 4): ", unaryRightAdd(1, 2, 3, 4));
31
32     std::cout << "Unary left and right fold with subtraction:"
33     << fmt::format("\n{}{}\n{}{}\n\n",
34                     "unaryLeftSubtract(1, 2, 3, 4): ",
35                     unaryLeftSubtract(1, 2, 3, 4),
36                     "unaryRightSubtract(1, 2, 3, 4): ",
37                     unaryRightSubtract(1, 2, 3, 4));
38 }
```

```
Unary left and right fold with addition:
unaryLeftAdd(1, 2, 3, 4): 10
unaryRightAdd(1, 2, 3, 4): 10
```

```
Unary left and right fold with subtraction:
unaryLeftSubtract(1, 2, 3, 4): -8
unaryRightSubtract(1, 2, 3, 4): -2
```

```
unaryLeftSubtract(1, 2, 3, 4)
```

```
unaryRightSubtract(1, 2, 3, 4)
```

```
1 // fig15_16.cpp
2 // Binary fold expressions.
3 #include <fmt/format.h>
4 #include <iostream>
5
6 template <typename... Items>
7 auto binaryLeftAdd(Items... items) {
8     return (0 + ... + items); // binary left fold
9 }
10
11 template <typename... Items>
12 auto binaryRightAdd(Items... items) {
13     return (items + ... + 0); // binary right fold
14 }
15
16 template <typename... Items>
17 auto binaryLeftSubtract(Items... items) {
18     return (0 - ... - items); // binary left fold
19 }
20
21 template <typename... Items>
22 auto binaryRightSubtract(Items... items) {
23     return (items - ... - 0); // binary right fold
24 }
25
```

```
26 int main() {
27     std::cout << "Binary left and right fold with addition:"
28     << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
29                     "binaryLeftAdd(): ", binaryLeftAdd(),
30                     "binaryLeftAdd(1, 2, 3, 4): ", binaryLeftAdd(1, 2, 3, 4),
31                     "binaryRightAdd(): ", binaryRightAdd(),
32                     "binaryRightAdd(1, 2, 3, 4): ", binaryRightAdd(1, 2, 3, 4));
33
34     std::cout << "Binary left and right fold with subtraction:"
35     << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
36                     "binaryLeftSubtract(): ", binaryLeftSubtract(),
37                     "binaryLeftSubtract(1, 2, 3, 4): ",
38                     binaryLeftSubtract(1, 2, 3, 4),
39                     "binaryRightSubtract(): ", binaryRightSubtract(),
40                     "binaryRightSubtract(1, 2, 3, 4): ",
41                     binaryRightSubtract(1, 2, 3, 4));
42 }
```

Binary left and right fold with addition:

```
binaryLeftAdd(): 0
binaryLeftAdd(1, 2, 3, 4): 10
binaryRightAdd(): 0
binaryRightAdd(1, 2, 3, 4): 10
```

Binary left and right fold with subtraction:

```
binaryLeftSubtract(): 0
binaryLeftSubtract(1, 2, 3, 4): -10
binaryRightSubtract(): 0
binaryRightSubtract(1, 2, 3, 4): -2
```

```
(std::cout << items << "\n")
```

```
1 // fig15_17.cpp
2 // Repeating a task using the comma operator and fold expressions.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <vector>
6
7 template <typename... Items>
8 void printItems(Items... items) {
9     ((std::cout << items << "\n"), ...); // binary left fold
10 }
11
12 int main() {
13     std::cout << "printItems(1, 2.2, \"hello\"):\\n";
14     printItems(1, 2.2, "hello");
15 }
```

```
printItems(1, 2.2, "hello"):
1
2.2
hello
```

```
1 // fig15_18.cpp
2 // Constraining a variadic-function-template parameter pack to
3 // elements of the same type.
4 #include <concepts>
5 #include <iostream>
6 #include <string>
7
8 template <typename T, typename... U>
9 concept SameTypeElements = (std::same_as<T, U> && ...);
10
11 // add one or more arguments with a fold expression
12 template <typename FirstItem, typename... RemainingItems>
13     requires SameTypeElements<FirstItem, RemainingItems...>
14 auto foldingSum(FirstItem first, RemainingItems... theRest) {
15     return (first + ... + theRest); // expand the parameter
16 }
17
18 int main() {
19     using namespace std::literals;
20
21     foldingSum(1, 2, 3); // valid: all are int values
22     foldingSum("s"s, "u"s, "m"s); // valid: all are std::string objects
23     foldingSum(1, 2.2, "hello"s); // error: three different types
24 }
```

```
fig15_18.cpp:23:4: error: no matching function for call to
foldingSum(1, 2.2, "hello"s); // error: three different types
^~~~~~
```

```
fig15_18.cpp:14:6: note: candidate template ignored: constraints not satisfied
[with FirstItem = int, RemainingItems = <double, std::basic_string<char>>]
auto foldingSum(FirstItem first, RemainingItems... theRest) {
    ^
```

```
fig15_18.cpp:13:13: note: because sic_string<char> >
requires SameTypeElements<FirstItem, RemainingItems...>
    ^
```

```
fig15_18.cpp:9:34: note: because false
concept SameTypeElements = (std::same_as<T, U> && ...);
^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/
concepts:63:19: note: because false
    = __detail::__same_as<Tp, Up> && __detail::__same_as<Up, Tp>;
           ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/
concepts:57:27: note: because false
concept __same_as = std::is_same_v<Tp, Up>;
           ^
fig15_18.cpp:9:34: note: and evaluated to false
concept SameTypeElements = (std::same_as<T, U> && ...);
^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/
concepts:63:19: note: because sic_string<char> >
    = __detail::__same_as<Tp, Up> && __detail::__same_as<Up, Tp>;
           ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/
concepts:57:27: note: because evaluated to false
concept __same_as = std::is_same_v<Tp, Up>;
           ^
```

1 error generated.

error: no matching function for call to

note: candidate template ignored: constraints not satisfied

<http://www.erwin-unruh.de/primorig.html>

```
| Type D<2>' ("primes.cpp",L2/C25).  
| Type D<3>' ("primes.cpp",L2/C25).  
| Type D<5>' ("primes.cpp",L2/C25).  
| Type D<7>' ("primes.cpp",L2/C25).
```

```
1 // fig15_19.cpp
2 // Calculating factorials at compile time.
3 #include <iostream>
4
5 // Factorial value metafunction calculates factorials recursively
6 template <int N>
7 struct Factorial {
8     static constexpr long long value{N * Factorial<N - 1>::value};
9 };
10
11 // Factorial specialization for the base case
12 template <>
13 struct Factorial<0> {
14     static constexpr long long value{1}; // 0! is 1
15 };
16
17 // constexpr compile-time recursive factorial
18 constexpr long long recursiveFactorial(int number) {
19     if (number <= 1) {
20         return 1; // base cases: 0! = 1 and 1! = 1
21     }
22     else { // recursion step
23         return number * recursiveFactorial(number - 1);
24     }
25 }
26
27 // constexpr compile-time iterative factorial
28 constexpr long long iterativeFactorial(int number) {
29     long long factorial{1}; // result for 0! and 1!
30
31     for (long long i{2}; i <= number; ++i) {
32         factorial *= i;
33     }
34 }
```

```
35     return factorial;
36 }
37
38 int main() {
39     // "calling" a value metafunction requires specializing
40     // the template and accessing its static value member
41     std::cout << "CALCULATING FACTORIALS AT COMPILE-TIME"
42         << "\nWITH A RECURSIVE METAFUNCTION"
43         << "\nFactorial(0): " << Factorial<0>::value
44         << "\nFactorial(1): " << Factorial<1>::value
45         << "\nFactorial(2): " << Factorial<2>::value
46         << "\nFactorial(3): " << Factorial<3>::value
47         << "\nFactorial(4): " << Factorial<4>::value
48         << "\nFactorial(5): " << Factorial<5>::value;
49
50     // calling the recursive constexpr function recursiveFactorial
51     std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE-TIME"
52         << "\nWITH A RECURSIVE CONSTEXPR FUNCTION"
53         << "\nrecursiveFactorial(0): " << recursiveFactorial(0)
54         << "\nrecursiveFactorial(1): " << recursiveFactorial(1)
55         << "\nrecursiveFactorial(2): " << recursiveFactorial(2)
56         << "\nrecursiveFactorial(3): " << recursiveFactorial(3)
57         << "\nrecursiveFactorial(4): " << recursiveFactorial(4)
58         << "\nrecursiveFactorial(5): " << recursiveFactorial(5);
59
60     // calling the iterative constexpr function recursiveFactorial
61     std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE-TIME"
62         << "\nWITH AN ITERATIVE CONSTEXPR FUNCTION"
63         << "\niterativeFactorial(0): " << iterativeFactorial(0)
64         << "\niterativeFactorial(1): " << iterativeFactorial(1)
65         << "\niterativeFactorial(2): " << iterativeFactorial(2)
66         << "\niterativeFactorial(3): " << iterativeFactorial(3)
67         << "\niterativeFactorial(4): " << iterativeFactorial(4)
68         << "\niterativeFactorial(5): " << iterativeFactorial(5) << "\n";
69 }
```

CALCULATING FACTORIALS AT COMPILE-TIME
WITH A RECURSIVE METAFUNCTION

```
Factorial(0): 1
Factorial(1): 1
Factorial(2): 2
Factorial(3): 6
Factorial(4): 24
Factorial(5): 120
```

CALCULATING FACTORIALS AT COMPILE-TIME
WITH A RECURSIVE CONSTEXPR FUNCTION

```
recursiveFactorial(0): 1
recursiveFactorial(1): 1
recursiveFactorial(2): 2
recursiveFactorial(3): 6
recursiveFactorial(4): 24
recursiveFactorial(5): 120
```

CALCULATING FACTORIALS AT COMPILE-TIME
WITH AN ITERATIVE CONSTEXPR FUNCTION

```
iterativeFactorial(0): 1
iterativeFactorial(1): 1
iterativeFactorial(2): 2
iterativeFactorial(3): 6
iterativeFactorial(4): 24
iterativeFactorial(5): 120
```

```
1 // fig15_20.cpp
2 // Implementing customDistance using template metaprogramming.
3 #include <array>
4 #include <iostream>
5 #include <iterator>
6 #include <list>
7 #include <ranges>
8 #include <type_traits>
9
10 // calculate the distance (number of items) between two iterators;
11 // requires at least input iterators
12 template <typename Iterator>
13     requires std::input_iterator<Iterator>
14 auto customDistance(Iterator begin, Iterator end) {
15     // for random-access iterators subtract the iterators
16     if constexpr (std::is_same<Iterator::iterator_category,
17         std::random_access_iterator_tag>::value) {
18
19         std::cout << "customDistance with random-access iterators\n";
20         return end - begin; // O(1) operation for random-access iterators
21     }
22     else { // for all other iterators
23         std::cout << "customDistance with non-random-access iterators\n";
24         std::ptrdiff_t count{0};
25
26         // increment from begin to end and count number of iterations;
27         // O(n) operation for non-random-access iterators
28         for (auto& iter{begin}; iter != end; ++iter) {
29             ++count;
30         }
31
32         return count;
33     }
34 }
35
36 int main() {
37     const std::array ints1{1, 2, 3, 4, 5}; // has random-access iterators
38     const std::list ints2{1, 2, 3}; // has bidirectional iterators
39
40     auto result1{customDistance(ints1.begin(), ints1.end())};
41     std::cout << "ints1 number of elements: " << result1 << "\n";
42     auto result2{customDistance(ints2.begin(), ints2.end())};
43     std::cout << "ints1 number of elements: " << result2 << "\n";
44 }
```

```
customDistance with random-access iterators
ints1 number of elements: 5
customDistance with non-random-access iterators
ints1 number of elements: 3
```

`std::random_access_iterator_tag`

```
1 // fig15_21.cpp
2 // Adding and removing type attributes with type metaprograms.
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <type_traits>
6
7 // add const to a type T
8 template <typename T>
9 struct my_add_const {
10     using type = const T;
11 };
12
13 // general case: no pointer in type, so set nested type variable to T
14 template <typename T>
15 struct my_remove_ptr {
16     using type = T;
17 };
18
19 // partial template specialization: T is a pointer type, so remove *
20 template <typename T>
21 struct my_remove_ptr<T*> {
22     using type = T;
23 };
24
25 int main() {
26     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}\n",
27         "ADD CONST TO A TYPE VIA A CUSTOM TYPE METAPROGRAM",
28         "std::is_same<const int, my_add_const<int>::type>::value: ",
29         std::is_same<const int, my_add_const<int>::type>::value,
30         "std::is_same<int* const, my_add_const<int*>::type>::value: ",
31         std::is_same<int* const, my_add_const<int*>::type>::value);
32
33     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}\n",
34         "REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAPROGRAM",
35         "std::is_same<int, my_remove_ptr<int>::type>::value: ",
36         std::is_same<int, my_remove_ptr<int>::type>::value,
37         "std::is_same<int, my_remove_ptr<int*>::type>::value: ",
38         std::is_same<int, my_remove_ptr<int*>::type>::value);
39 }
```

```

40     std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
41         "ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS",
42         "std::is_same<int&, std::add_lvalue_reference<int>::type>::value: ",
43         std::is_same<int&, std::add_lvalue_reference<int>::type>::value,
44         "std::is_same<int&&, std::add_rvalue_reference<int>::type>::value: ",
45         std::is_same<int&&, std::add_rvalue_reference<int>::type>::value);
46
47     std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n",
48         "REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS",
49         "std::is_same<int, std::remove_reference<int>::type>::value: ",
50         std::is_same<int, std::remove_reference<int>::type>::value,
51         "std::is_same<int, std::remove_reference<int&>::type>::value: ",
52         std::is_same<int, std::remove_reference<int&>::type>::value,
53         "std::is_same<int, std::remove_reference<int&&>::type>::value: ",
54         std::is_same<int, std::remove_reference<int&&>::type>::value);
55 }

```

ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION

```

std::is_same<const int, my_add_const<int>::type>::value: true
std::is_same<int* const, my_add_const<int*>::type>::value: true

```

REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION

```

std::is_same<int, my_remove_ptr<int>::type>::value: true
std::is_same<int, my_remove_ptr<int*>::type>::value: true

```

ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS

```

std::is_same<int&, std::add_lvalue_reference<int>::type>::value: true
std::is_same<int&&, std::add_rvalue_reference<int>::type>::value: true

```

REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS

```

std::is_same<int, std::remove_reference<int>::type>::value: true
std::is_same<int, std::remove_reference<int&>::type>::value: true
std::is_same<int, std::remove_reference<int&&>::type>::value: true

```

```
docker pull teeks99/clang-ubuntu:13
```

<https://docs.docker.com/get-started/overview/>

```
1 // fig16_01.cpp
2 // Importing a standard library header as a header unit.
3 import <iostream>; // instead of #include <iostream>
4
5 int main() {
6     std::cout << "Welcome to C++20 Modules!\n";
7 }
```

Welcome to C++20 Modules!

<https://visualstudio.microsoft.com/vs/preview/>

```
g++ -fmodules-ts -x c++-system-header iostream
```

```
g++ -fmodules-ts fig16_01.cpp -o fig16_01
```

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps fig16_01.cpp -o fig16_01
```

```
1 // Fig. 16.2: welcome.ixx
2 // Primary module interface for a module named welcome.
3 export module welcome; // introduces the module name
4
5 import <string>; // class string is used in this module
6
7 // exporting a function
8 export std::string welcomeStandalone() {
9     return "welcomeStandalone function called";
10 }
11
12 // export block exports all items in the block's braces
13 export {
14     std::string welcomeFromExportBlock() {
15         return "welcomeFromExportBlock function called";
16     }
17 }
18
19 // exporting a namespace exports all items in the namespace
20 export namespace TestNamespace1 {
21     std::string welcomeFromTestNamespace1() {
22         return "welcomeFromTestNamespace1 function called";
23     }
24 }
25
26 // exporting an item in a namespace exports the namespace name too
27 namespace TestNamespace2 {
28     export std::string welcomeFromTestNamespace2() {
29         return "welcomeFromTestNamespace2 function called";
30     }
31 }
```

```
namespace std {  
    // standard library header's declarations  
}
```

```
TestNamespace1::welcomeInTestNamespace1()
```

```
using TestNamespace1::welcomeInTestNamespace1;
```

```
using namespace TestNamespace1;
```

```
1 // fig16_03.cpp
2 // Importing a module and using its exported items.
3 import <iostream>;
4 import welcome; // import the welcome module
5
6 int main() {
7     std::cout << welcomeStandalone() << '\n'
8         << welcomeFromExportBlock() << '\n'
9         << TestNamespace1::welcomeFromTestNamespace1() << '\n'
10        << TestNamespace2::welcomeFromTestNamespace2() << '\n';
11 }
```

welcomeStandalone function called
welcomeFromExportBlock function called
welcomeFromTestNamespace1 function called
welcomeFromTestNamespace2 function called

```
g++ -fmodules-ts -x c++-system-header string  
g++ -fmodules-ts -x c++-system-header iostream
```

```
g++ -fmodules-ts -c -x c++ welcome..hxx
```

```
g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -xc++ welcome..hxx  
-Xclang -emit-module-interface -o welcome.pcm
```

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fprebuilt-module-path=.  
fig16_03.cpp welcome.pcm -o fig16_03
```

```
1 // Fig. 16.4: deitel.math.ixx
2 // Primary module interface for a module named deitel.math.
3 export module deitel.math; // introduces the module name
4
5 namespace deitel::math {
6     // exported function square; namespace deitel::math implicitly exported
7     export int square(int x) {
8         return x * x;
9     }
10
11    // non-exported function cube is not implicitly exported
12    int cube(int x) {
13        return x * x * x;
14    }
15};
```

```
1 // fig16_05.cpp
2 // Showing that a module's non-exported identifiers are inaccessible.
3 import <iostream>;
4 import deitel.math; // import the welcome module
5
6 int main() {
7     // can call square because it's exported from namespace deitel::math,
8     // which implicitly exports the namespace
9     std::cout << "square(3) = " << deitel::math::square(3) << '\n';
10
11    // cannot call cube because it's not exported
12    std::cout << "cube(3) = " << deitel::math::cube(3) << '\n';
13 }
```

```
Build started...
1>----- Build started: Project: modules_demo, Configuration: Debug Win32 -----
---  
1>Scanning sources for module dependencies...
1>deitel.math.ixx
1>fig16_05.cpp
1>Compiling...
1>deitel.math.ixx
1>fig16_05.cpp

1>C:\Users\pauldeitel\Documents\examples\ch16\fig16_04-
05\fig16_05.cpp(12,47): error C2039: 'cube': is not a member of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-05\de-
itel.math.ixx(5): message : see declaration of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-
05\fig16_05.cpp(12,51): error C3861: 'cube': identifier not found

1>Done building project "modules_demo.vcxproj" -- FAILED.
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

```
fig16_05.cpp: In function 'int main()':  
fig16_05.cpp:12:49: error: 'cube' is not a member of 'deitel::math'  
12 |     std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';  
|           ^~~~
```

```
-fimplicit-module-maps -xc++ deitel.math..hxx  
-Xclang -emit-module-interface -o deitel.math.pcm
```

```
-fimplicit-module-maps -fprebuilt-module-path=.  
fig16_05.cpp deitel.math.pcm -o fig16_05
```

```
1 // Fig. 16.6: deitel.math.ixx
2 // Primary module interface for a module named deitel.math
3 // with a :private module fragment.
4 export module deitel.math; // introduces the module name
5
6 import <numeric>;
7 import <vector>;
8
9 export namespace deitel::math {
10     // calculate the average of a vector<int>'s elements
11     double average(const std::vector<int>& values);
12 };
13
14 module :private; // private implementation details
15
16 namespace deitel::math {
17     // average function's implementation
18     double average(const std::vector<int>& values) {
19         double total{std::accumulate(values.begin(), values.end(), 0.0)};
20         return total / values.size();
21     }
22 };
```

```
1 // fig16_07.cpp
2 // Using the deitel.math module's average function.
3 import <algorithm>;
4 import <iostream>;
5 import <iterator>;
6 import <vector>;
7 import deitel.math; // import the deitel.math module
8
9 int main() {
10     std::ostream_iterator<int> output(std::cout, " ");
11     std::vector integers{1, 2, 3, 4};
12
13     std::cout << "vector integers: ";
14     std::copy(integers.begin(), integers.end(), output);
15
16     std::cout << "\naverage of integer's elements: "
17         << deitel::math::average(integers) << '\n';
18 }
```

```
vector integers: 1 2 3 4
average of integer's elements: 2.5
```

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

```
g++ -fmodules-ts fig16_07.cpp deitel.math.o -o fig16_07
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -xc++ deitel.math.ixx  
-Xclang -emit-module-interface -o deitel.math.pcm
```

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fprebuilt-module-path=.  
fig16_07.cpp deitel.math.pcm -o fig16_07
```

```
1 // Fig. 16.8: deitel.math.ixx
2 // Primary module interface for a module named deitel.math.
3 export module deitel.math; // introduces the module name
4
5 import <vector>;
6
7 export namespace deitel::math {
8     // calculate the average of a vector<int>'s elements
9     double average(const std::vector<int>& values);
10}
```

```
1 // Fig. 16.9: deitel.math-impl.cpp
2 // Module implementation unit for the module deitel.math.
3 module deitel.math; // this file's contents belong to module deitel.math
4
5 import <numeric>;
6 import <vector>;
7
8 namespace deitel::math {
9     // average function's implementation
10    double average(const std::vector<int>& values) {
11        double total{std::accumulate(values.begin(), values.end(), 0.0)};
12        return total / values.size();
13    }
14};
```

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

```
g++ -fmodules-ts -c deitel.math-impl.cpp
```

```
g++ -fmodules-ts fig16_07.cpp deitel.math.o deitel.math-impl.o  
-o fig16_07
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -xc++ deitel.math.ixx  
-Xclang -emit-module-interface -o deitel.math.pcm
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fmodule-file=deitel.math.pcm  
deitel.math-impl.cpp
```

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fprebuilt-module-path=. fig16_07.cpp  
deitel.math-impl.o deitel.math.pcm -o fig16_07
```

```
1 // Fig. 16.10: deitel.time.ixx
2 // Primary module interface for a simple Time class.
3 export module deitel.time; // declare primary module interface
4
5 import <string>; // rather than #include <string>
6
7 export namespace deitel::time {
8     class Time {
9         public:
10             // default constructor because it can be called with no arguments
11             explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13             std::string toString() const;
14         private:
15             int m_hour{0}; // 0 - 23 (24-hour clock format)
16             int m_minute{0}; // 0 - 59
17             int m_second{0}; // 0 - 59
18     };
19 }
```

```
using namespace deitel::time;
```

```
1 // Fig. 16.11: deitel.time-impl.cpp
2 // deitel.time module implementation unit containing the
3 // Time class member function definitions.
4 module deitel.time; // module implementation unit for deitel.time
5
6 import <stdexcept>;
7 import <string>;
8 using namespace deitel::time;
9
10 // Time constructor initializes each data member
11 Time::Time(int hour, int minute, int second) {
12     // validate hour, minute and second
13     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
14         (second < 0 || second >= 60)) {
15         throw std::invalid_argument{
16             "hour, minute or second was out of range"};
17     }
18
19     m_hour = hour;
20     m_minute = minute;
21     m_second = second;
22 }
23
24 // return a string representation of the Time
25 std::string Time::toString() const {
26     using namespace std::string_literals;
27
28     return "Hour: "s + std::to_string(m_hour) +
29             "\nMinute: "s + std::to_string(m_minute) +
30             "\nSecond: "s + std::to_string(m_second);
31 }
```

```
1 // fig16_12.cpp
2 // Importing the deitel.time module and using the modularized Time class.
3 import <iostream>;
4 import <stdexcept>;
5 import <string>;
6
7 import deitel.time;
8 using namespace deitel::time;
9
10 int main() {
11     const Time t{12, 25, 42}; // hour, minute and second specified
12
13     std::cout << "Time t:\n" << t.toString() << "\n\n";
14
15     // attempt to initialize t2 with invalid values
16     try {
17         const Time t2{27, 74, 99}; // all bad values specified
18     }
19     catch (const std::invalid_argument& e) {
20         std::cout << "t2 not created: " << e.what() << '\n';
21     }
22 }
```

Time t:
Hour: 12
Minute: 25
Second: 42

t2 not created: hour, minute or second was out of range

```
g++ -fmodules-ts -c -x c++ deitel.time..hxx
```

```
g++ -fmodules-ts -c -x c++ deitel.time-impl..hxx
```

```
g++ -fmodules-ts fig16_12.cpp deitel.time.o deitel.time-impl.o  
-o fig16_07
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -xc++ deitel.time..hxx  
-Xclang -emit-module-interface -o deitel.time.pcm
```

```
clang++-13 -c -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fmodule-file=deitel.time.pcm  
deitel.time-impl.cpp
```

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps -fprebuilt-module-path=.  
fig16_12.cpp deitel.time-impl.o deitel.time.pcm -o fig16_12
```

```
1 // Fig. 16.13: deitel.math-powers.ixx
2 // Module interface partition unit deitel.math:powers.
3 export module deitel.math:powers;
4
5 export namespace deitel::math {
6     double square(double x) { return x * x; }
7     double cube(double x) { return x * x * x; }
8 }
```

```
1 // Fig. 16.14: deitel.math-roots.ixx
2 // Module interface partition unit deitel.math:roots.
3 export module deitel.math:roots;
4
5 import <cmath>;
6
7 export namespace deitel::math {
8     double squareRoot(double x) { return std::sqrt(x); }
9     double cubeRoot(double x) { return std::cbrt(x); }
10 }
```

```
1 // Fig. 16.15: deitel.math.ixx
2 // Primary module interface unit deitel.math exports declarations from
3 // the module interface partitions :powers and :roots.
4 export module deitel.math; // declares the primary module interface unit
5
6 // import and re-export the declarations in the module
7 // interface partitions :powers and :roots
8 export import :powers;
9 export import :roots;
```

```
1 // fig16_16.cpp
2 // Using the deitel.math module's functions.
3 import <iostream>;
4 import deitel.math; // import the deitel.math module
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5)
11    << "\nsquareRoot(9): " << squareRoot(9)
12    << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
13 }
```

```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

```
g++ -fmodules-ts -c -x c++ deitel.math-powers..hxx  
g++ -fmodules-ts -c -x c++ deitel.math-roots..hxx
```

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

```
g++ -fmodules-ts -I ../../libraries/fmt/include fig16_16.cpp  
../../libraries/fmt/src/format.cc deitel.math-powers.o  
deitel.math-roots.o deitel.math.o -o fig16_16
```

```
export module deitel.math:powers;
```

```
export module deitel.math.powers;
```

```
1 // Fig. 16.17: deitel.math.powers.ixx
2 // Primary module interface unit deitel.math.powers.
3 export module deitel.math.powers;
4
5 export namespace deitel::math {
6     double square(double x) { return x * x; }
7     double cube(double x) { return x * x * x; }
8 }
```

```
1 // fig16_18.cpp
2 // Using the deitel.math.powers module's functions.
3 import <iostream>;
4 import deitel.math.powers; // import the deitel.math.powers module
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5) << '\n';
11 }
```

```
square(6): 36
cube(5): 125
```

```
export module deitel.math:roots;
```

```
export module deitel.math.roots;
```

```
1 // Fig. 16.19: deitel.math.roots.ixx
2 // Primary module interface unit deitel.math.roots.
3 export module deitel.math.roots;
4
5 import <cmath>;
6
7 export namespace deitel::math {
8     double squareRoot(double x) { return std::sqrt(x); }
9     double cubeRoot(double x) { return std::cbrt(x); }
10 }
```

```
1 // fig16_20.cpp
2 // Using the deitel.math.roots module's functions.
3 import <iostream>;
4 import deitel.math.roots; // import the deitel.math.roots module
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "squareRoot(9): " << squareRoot(9)
10      << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
11 }
```

```
square(6): 36
cube(5): 125
```

```
1 // Fig. 16.21: deitel.math.ixx
2 // Primary module interface unit deitel.math aggregates declarations
3 // from "submodules" deitel.math.powers and deitel.math.roots.
4 export module deitel.math; // primary module interface unit
5
6 // import and re-export deitel.math.powers and deitel.math.roots
7 export import deitel.math.powers;
8 export import deitel.math.roots;
```

```
1 // fig16_22.cpp
2 // Using the deitel.math module's functions.
3 import <iostream>;
4 import deitel.math; // import the deitel.math module
5
6 using namespace deitel::math;
7
8 int main() {
9     std::cout << "square(6): " << square(6)
10    << "\ncube(5): " << cube(5)
11    << "\nsquareRoot(9): " << squareRoot(9)
12    << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
13 }
```

```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

```
1 // Fig. 16.23.cpp
2 // Importing Microsoft's modularized standard library.
3 import std.core; // provides access to most of the C++ standard library
4
5 int main() {
6     std::cout << "Welcome to C++20 Modules!\n";
7 }
```

Welcome to C++20 Modules!

```
clang++-13 -std=c++20 -stdlib=libc++ -fimplicit-modules  
-fimplicit-module-maps fig16_23.cpp -o fig16_23
```

```
1 // Fig. 16.24: moduleA.ixx
2 // Primary module interface unit that imports moduleB.
3 export module moduleA; // declares the primary module interface unit
4
5 export import moduleB; // import and re-export moduleB
```

```
1 // Fig. 16.25: moduleB.ixx
2 // Primary module interface unit that imports moduleA.
3 export module moduleB; // declares the primary module interface unit
4
5 export import moduleA; // import and re-export moduleA
```

error : Cannot build the following source files because there is a cyclic dependency between them: ch16\fig16_24-26\moduleA.ixx depends on ch16\fig16_24-26\moduleB.ixx depends on ch16\fig16_24-26\moduleA.ixx.

```
1 // Fig. 16.26: moduleA.ixx
2 // Primary module interface unit that exports function cube.
3 export module moduleA; // declares the primary module interface unit
4
5 export int cube(int x) { return x * x * x; }
```

```
1 // Fig. 16.27: moduleB.ixx
2 // Primary module interface unit that imports, but does not export,
3 // moduleA and exports function square.
4 export module moduleB; // declares the primary module interface unit
5
6 import moduleA; // import but do not export moduleA
7
8 export int square(int x) { return x * x; }
```

```
1 // fig16_28.cpp
2 // Showing that moduleB does not implicitly export moduleA's function.
3 import <iostream>;
4 import moduleB;
5
6 int main() {
7     std::cout << "square(6): " << square(6) // exported from moduleB
8         << "\ncube(5): " << cube(5) << '\n'; // not exported from moduleB
9 }
```

```
1 // Fig. 16.29: deitel.time.ixx
2 // Primary module interface unit for the deitel.time module.
3 export module deitel.time; // declare the primary module interface
4
5 import <string>; // rather than #include <string>
6
7 namespace deitel::time {
8     class Time { // not exported
9         public:
10             // default constructor because it can be called with no arguments
11             explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13             std::string toString() const;
14         private:
15             int m_hour{0}; // 0 - 23 (24-hour clock format)
16             int m_minute{0}; // 0 - 59
17             int m_second{0}; // 0 - 59
18     };
19
20     // exported function returns a valid Time
21     export Time getTime() { return Time(6, 45, 0); }
22 }
```

error C2065: 'Time': undeclared identifier

```
1 // fig16_30.cpp
2 // Showing that type deitel::time::Time is reachable
3 // and its public members are visible.
4 import <iostream>;
5 import deitel.time;
6
7 int main() {
8     // initialize t with the object returned by getTime; cannot declare t
9     // as type Time because the type is not exported, and thus not visible
10    auto t{ deitel::time::getTime() };
11
12    // Time's toString function is reachable, even though
13    // class Time was not exported by module deitel.time
14    std::cout << "Time t:\n" << t.toString() << "\n\n";
15 }
```

```
Time t:
Hour: 6
Minute: 45
Second: 0
```

```
#include <iostream>

int main() {
    std::cout << "Welcome to C++20 Modules!\\n";
}
```

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