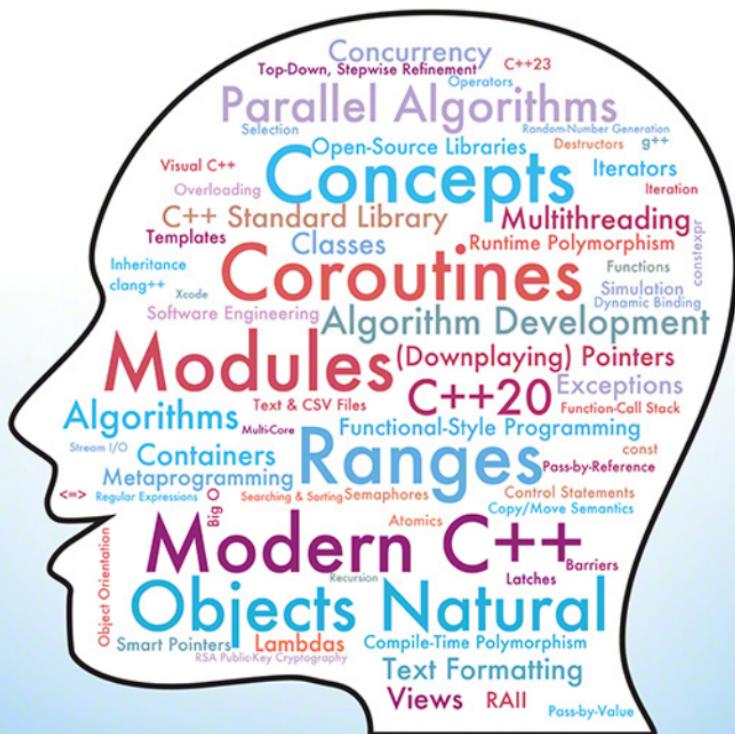




C++

HOW TO PROGRAM An Objects-Natural Approach



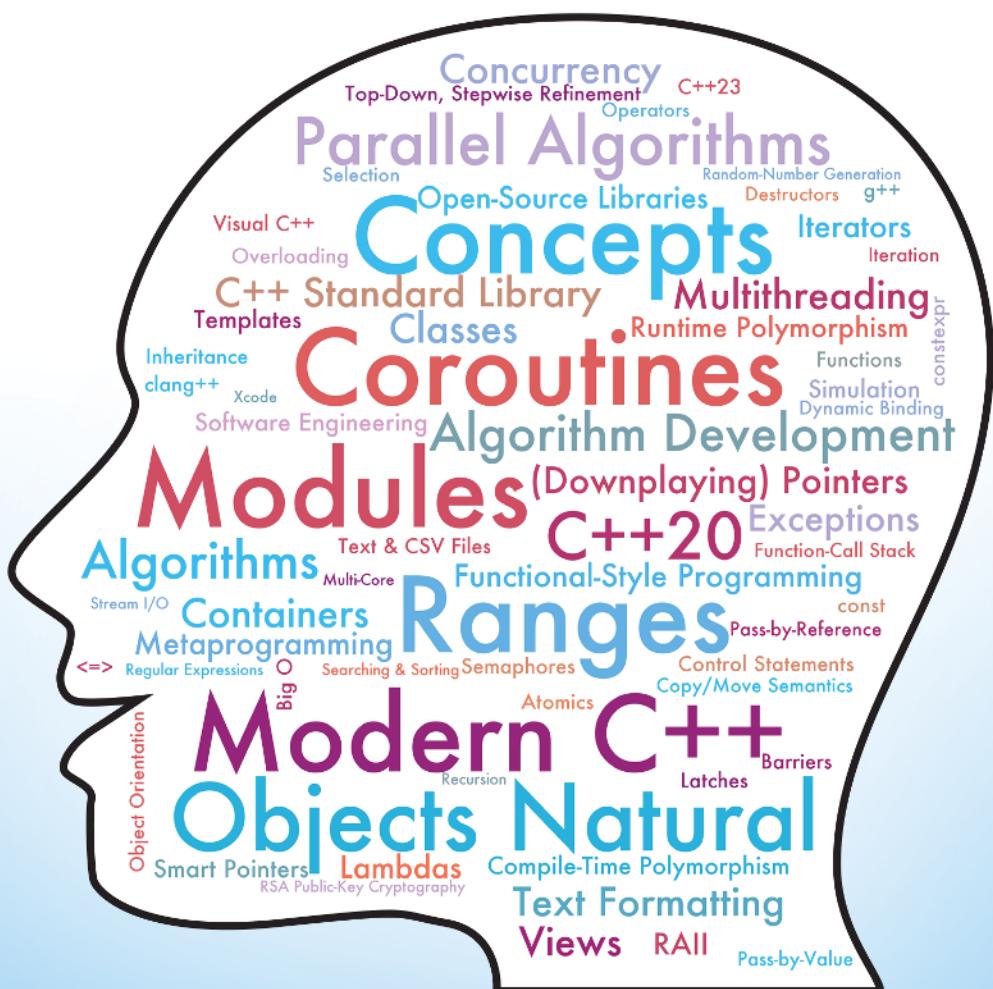
PAUL DEITEL
HARVEY DEITEL

11
ELEVENTH
EDITION



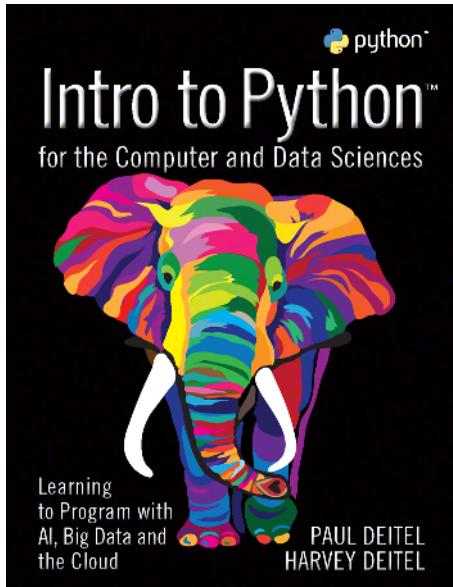
HOW TO PROGRAM

An Objects-Natural Approach

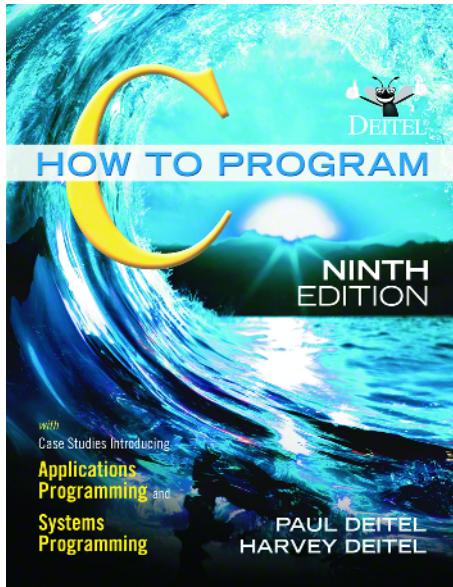


11
ELEVENTH
EDITION

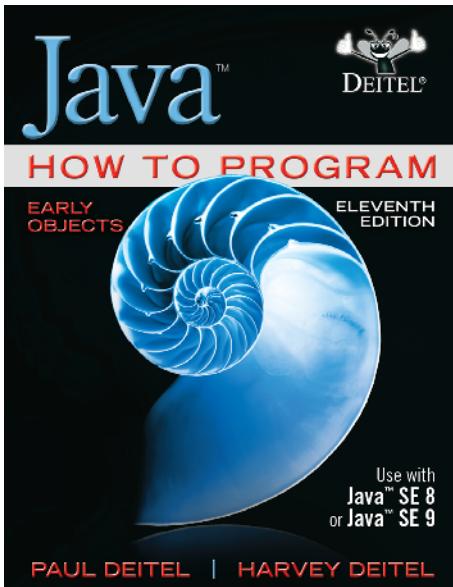
Recent Deitel Textbooks



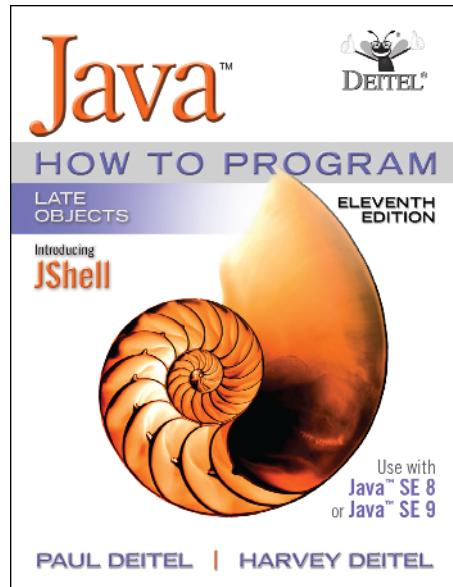
<https://deitel.com/pycds>



<https://deitel.com/cht9>



<https://deitel.com/jhtp11EOV>



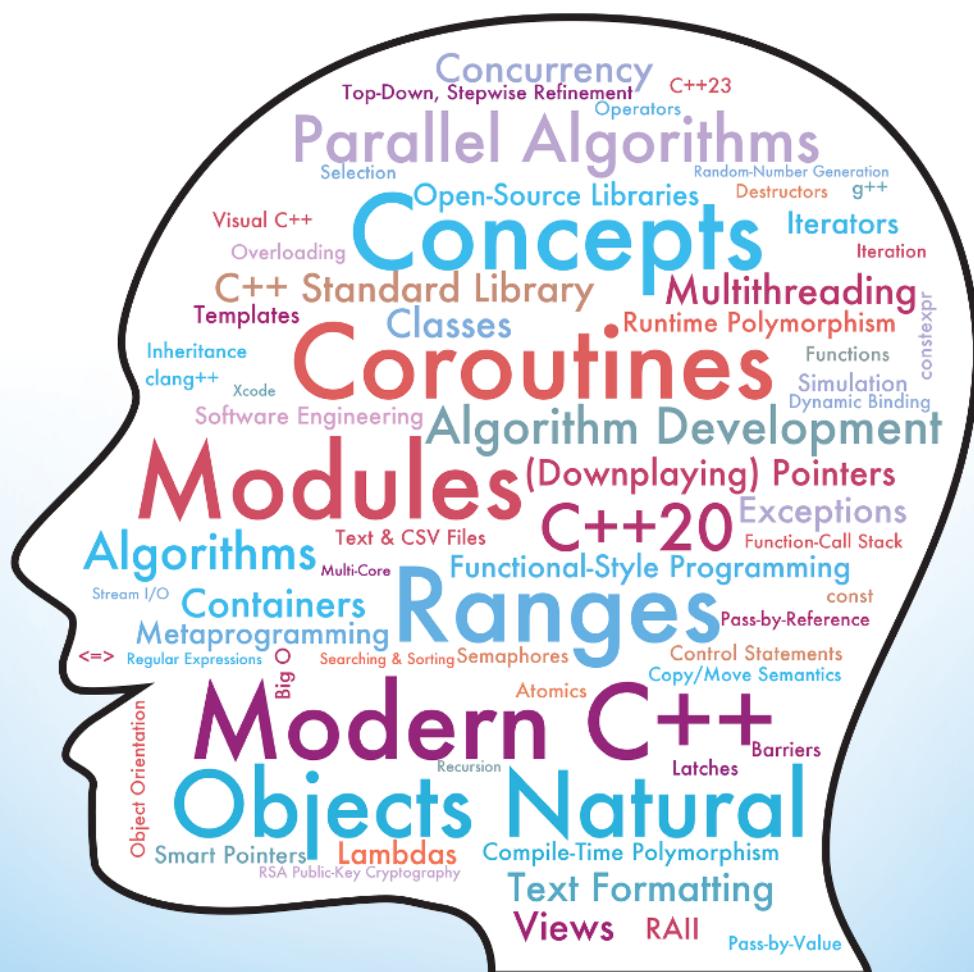
<https://deitel.com/jhtp11LOV>



PAUL DEITEL
Deitel & Associates, Inc.
HARVEY DEITEL
Deitel & Associates, Inc.

HOW TO PROGRAM

An Objects-Natural Approach



Pearson

11
ELEVENTH
EDITION

Content Management: Tracy Johnson

Content Production: Bob Engelhardt, Abhijeet Gope, K Madhusudhan

Product Marketing: Krista Clark

Rights and Permissions: Anjali Singh

Cover Designer: Paul Deitel, Harvey Deitel, Chuti Prasertsith

Cover Art: Paul Deitel (produced using Andreas Müller's open-source Python library `word_cld`—https://github.com/amueller/word_cld)

Please contact <https://support.pearson.com/getsupport/s/> with any queries on this content.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Copyright © 2024 by Pearson Education, Inc. or its affiliates, 221 River Street, Hoboken, NJ 07030. All Rights Reserved. Manufactured in the United States of America. This publication is protected by copyright, and permission should 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 otherwise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights and Permissions department, please visit www.pearsoned.com/permissions/.

Acknowledgments of third-party content appear on the appropriate page within the text.

PEARSON and REVEL are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks, logos, or icons that may appear in this work are the property of their respective owners, and any references to third-party trademarks, logos, icons, or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc., or its affiliates, authors, licensees, or distributors.

Deitel and the double-thumbs-up bug are registered trademarks of Deitel and Associates, Inc.

Library of Congress Cataloging-in-Publication Data

On file



uPDF
ISBN-10: 0-13-809242-7
ISBN-13: 978-0-13809242-9

ePub
ISBN-10: 0-13-809236-2
ISBN-13: 978-0-13809236-8

*In memory of Gordon Moore, co-founder of Intel
and father of Moore's Law:*

*For a career devoted to improving people's
lives through technology, innovation, industry,
entrepreneurship and philanthropy.*

Paul and Harvey Deitel

Pearson's Commitment to Diversity, Equity, and Inclusion

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

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

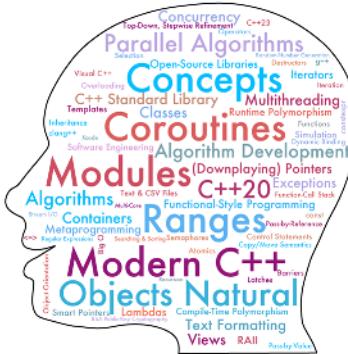
Our ambition is to purposefully contribute to a world where:

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

We are also committed to providing products that are fully accessible to all learners. As per Pearson's guidelines for accessible educational Web media, we test and retest the capabilities of our products against the highest standards for every release, following the WCAG guidelines in developing new products for copyright year 2022 and beyond. You can learn more about Pearson's commitment to accessibility at <https://www.pearson.com/us/accessibility.html>.

While we work hard to present unbiased, fully accessible content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.
- For accessibility-related issues, such as using assistive technology with Pearson products, alternative text requests, or accessibility documentation, email the Pearson Disability Support team at disability.support@pearson.com.



Contents

Preface

xxiii

Before You Begin

liii

I	Intro to Computers and C++	I
1.1	Introduction	2
1.2	Hardware	4
	1.2.1 Computer Organization	4
	1.2.2 Moore’s Law, Multi-Core Processors and Concurrent Programming	7
1.3	Data Hierarchies	10
1.4	Machine Languages, Assembly Languages and High-Level Languages	13
1.5	Operating Systems	14
1.6	C and C++	18
1.7	C++ Standard Library and Open-Source Libraries	19
1.8	Other Popular Programming Languages	21
1.9	Introduction to Object Orientation	23
1.10	Simplified View of a C++ Development Environment	25
1.11	Test-Driving a C++20 Application Various Ways	28
	1.11.1 Compiling and Running on Windows with Visual Studio Community Edition	29
	1.11.2 Compiling and Running with GNU C++ on Linux	32
	1.11.3 Compiling and Running with g++ in the GCC Docker Container	34
	1.11.4 Compiling and Running with clang++ in a Docker Container	36
	1.11.5 Compiling and Running with Xcode on macOS	37
1.12	Internet, World Wide Web, the Cloud and IoT	41
	1.12.1 The Internet: A Network of Networks	41
	1.12.2 The World Wide Web: Making the Internet User-Friendly	42
	1.12.3 The Cloud	42
	1.12.4 The Internet of Things (IoT)	42
	1.12.5 Edge Computing	43
	1.12.6 Mashups	43
1.13	Metaverse	44
	1.13.1 Virtual Reality (VR)	44
	1.13.2 Augmented Reality (AR)	45
	1.13.3 Mixed Reality	46
	1.13.4 Blockchain	46
	1.13.5 Bitcoin and Cryptocurrency	46

1.13.6	Ethereum	47
1.13.7	Non-Fungible Tokens (NFTs)	47
1.13.8	Web3	47
1.14	Software Development Technologies	48
1.15	How Big Is Big Data?	49
1.15.1	Big-Data Analytics	52
1.15.2	Data Science and Big Data Are Making a Difference: Use Cases	53
1.16	AI—at the Intersection of Computer Science and Data Science	54
1.16.1	Artificial Intelligence (AI)	54
1.16.2	Artificial General Intelligence (AGI)	55
1.16.3	Artificial Intelligence Milestones	55
1.16.4	Machine Learning	56
1.16.5	Deep Learning	56
1.16.6	Reinforcement Learning	57
1.16.7	Generative AI—ChatGPT and Dall-E 2	57
1.17	Wrap-Up	59

2 [Intro to C++20 Programming](#) 65

2.1	Introduction	66
2.2	First Program in C++: Displaying a Line of Text	66
2.3	Modifying Our First C++ Program	70
2.4	Another C++ Program: Adding Integers	71
2.5	Memory Concepts	75
2.6	Arithmetic	76
2.7	Decision Making: Equality and Relational Operators	79
2.8	Objects Natural Case Study:Creating and Using Objects of Standard-Library Class <code>string</code>	83
2.9	Wrap-Up	86

3 [Algorithm Development and Control Statements: Part I](#) 93

3.1	Introduction	94
3.2	Algorithms	95
3.3	Pseudocode	95
3.4	Control Structures	96
3.4.1	Sequence Structure	97
3.4.2	Selection Statements	98
3.4.3	Iteration Statements	98
3.4.4	Summary of Control Statements	99
3.5	<code>if</code> Single-Selection Statement	100
3.6	<code>if...else</code> Double-Selection Statement	101
3.6.1	Nested <code>if...else</code> Statements	102
3.6.2	Blocks	104
3.6.3	Conditional Operator (<code>?:</code>)	104
3.7	<code>while</code> Iteration Statement	105

3.8	Formulating Algorithms: Counter-Controlled Iteration	107
3.8.1	Pseudocode Algorithm with Counter-Controlled Iteration	107
3.8.2	Implementing Counter-Controlled Iteration	108
3.8.3	Integer Division and Truncation	110
3.8.4	Arithmetic Overflow	110
3.8.5	Input Validation	110
3.9	Formulating Algorithms: Sentinel-Controlled Iteration	111
3.9.1	Top-Down, Stepwise Refinement: The Top and First Refinement	112
3.9.2	Proceeding to the Second Refinement	112
3.9.3	Implementing Sentinel-Controlled Iteration	114
3.9.4	Mixed-Type Expressions and Implicit Type Promotions	116
3.9.5	Formatting Floating-Point Numbers	117
3.10	Formulating Algorithms: Nested Control Statements	118
3.10.1	Problem Statement	118
3.10.2	Top-Down, Stepwise Refinement: Pseudocode Representation of the Top	119
3.10.3	Top-Down, Stepwise Refinement: First Refinement	119
3.10.4	Top-Down, Stepwise Refinement: Second Refinement	120
3.10.5	Complete Second Refinement of the Pseudocode	120
3.10.6	Implementing the Program	121
3.10.7	Preventing Narrowing Conversions with Braced Initialization	123
3.11	Compound Assignment Operators	125
3.12	Increment and Decrement Operators	125
3.13	Fundamental Types Are Not Portable	128
3.14	Objects Natural Case Study: Super-Sized Integers	129
3.15	Wrap-Up	134

4	Control Statements, Part 2	145
4.1	Introduction	146
4.2	Essentials of Counter-Controlled Iteration	146
4.3	for Iteration Statement	148
4.4	Examples Using the for Statement	151
4.5	Application: Summing Even Integers; Introducing C++20 Text Formatting	152
4.6	Application: Compound-Interest Calculations; Introducing Format Specifiers	153
4.7	do...while Iteration Statement	158
4.8	switch Multiple-Selection Statement	159
4.9	Selection Statements with Initializers	165
4.10	break and continue Statements	167
4.11	Logical Operators	169
4.11.1	Logical AND (&&) Operator	169
4.11.2	Logical OR () Operator	170
4.11.3	Short-Circuit Evaluation	170
4.11.4	Logical Negation (!) Operator	171
4.11.5	Example: Producing Logical-Operator Truth Tables	171

x [Contents](#)

4.12	Confusing the Equality (==) and Assignment (=) Operators	173
4.13	Structured-Programming Summary	175
4.14	Objects Natural Case Study: Precise Monetary Calculations with the Boost Multiprecision Library	180
4.15	Wrap-Up	183

5 Functions and an Intro to Function Templates 189

5.1	Introduction	190
5.2	C++ Program Components	191
5.3	Math Library Functions	192
5.4	Function Definitions and Function Prototypes	194
5.5	Order of Evaluation of a Function's Arguments	197
5.6	Function-Prototype and Argument-Coercion Notes	197
5.6.1	Function Signatures and Function Prototypes	198
5.6.2	Argument Coercion	198
5.6.3	Argument-Promotion Rules and Implicit Conversions	198
5.7	C++ Standard Library Headers	200
5.8	Case Study: Random-Number Generation	203
5.8.1	Rolling a Six-Sided Die	204
5.8.2	Rolling a Six-Sided Die 60,000,000 Times	205
5.8.3	Seeding the Random-Number Generator	207
5.8.4	Seeding the Random-Number Generator with <code>random_device</code>	208
5.9	Case Study: Game of Chance; Introducing Scoped <code>enums</code>	209
5.10	Function-Call Stack and Activation Records	214
5.11	Inline Functions	217
5.12	References and Reference Parameters	219
5.13	Default Arguments	222
5.14	Function Overloading	224
5.15	Function Templates	227
5.16	Recursion	229
5.16.1	Factorials	230
5.16.2	Recursive Factorial Example	230
5.16.3	Recursive Fibonacci Series Example	234
5.16.4	Recursion vs. Iteration	237
5.17	Scope Rules	239
5.18	Unary Scope Resolution Operator	244
5.19	Lnfylun Lhqtmoh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz	245
5.20	Wrap-Up	248

6 arrays, vectors, Ranges and Functional-Style Programming 259

6.1	Introduction	260
6.2	arrays	261
6.3	Declaring arrays	262
6.4	Initializing array Elements in a Loop	262

6.5	Initializing an array with an Initializer List	265
6.6	Range-Based for Statement	267
6.7	Calculating array Element Values and an Intro to <code>constexpr</code>	269
6.8	Totaling array Elements	271
6.9	Using a Primitive Bar Chart to Display array Data Graphically	272
6.10	Using array Elements as Counters	274
6.11	Using arrays to Summarize Survey Results	275
6.12	Sorting and Searching arrays	277
6.13	Multidimensional arrays	279
6.14	Intro to Functional-Style Programming	283
6.14.1	What vs. How	283
6.14.2	Passing Functions as Arguments to Other Functions: Introducing Lambda Expressions	284
6.14.3	Filter, Map and Reduce: Intro to C++20's Ranges Library	286
6.15	Objects-Natural Case Study: C++ Standard Library	
	Class Template <code>vector</code>	290
6.16	Wrap-Up	297

7 (Downplaying) Pointers in Modern C++ **309**

7.1	Introduction	310
7.2	Pointer Variable Declarations and Initialization	312
7.2.1	Declaring Pointers	312
7.2.2	Initializing Pointers	312
7.2.3	Null Pointers	312
7.3	Pointer Operators	313
7.3.1	Address (&) Operator	313
7.3.2	Indirection (*) Operator	314
7.3.3	Using the Address (&) and Indirection (*) Operators	315
7.4	Pass-by-Reference with Pointers	316
7.5	Built-In Arrays	320
7.5.1	Declaring and Accessing a Built-In Array	320
7.5.2	Initializing Built-In Arrays	321
7.5.3	Passing Built-In Arrays to Functions	322
7.5.4	Declaring Built-In Array Parameters	322
7.5.5	Standard Library Functions <code>begin</code> and <code>end</code>	323
7.5.6	Built-In Array Limitations	323
7.6	Using C++20 <code>to_array</code> to Convert a Built-in Array to a <code>std::array</code>	324
7.7	Using <code>const</code> with Pointers and the Data Pointed To	325
7.7.1	Using a Nonconstant Pointer to Nonconstant Data	326
7.7.2	Using a Nonconstant Pointer to Constant Data	326
7.7.3	Using a Constant Pointer to Nonconstant Data	327
7.7.4	Using a Constant Pointer to Constant Data	327
7.8	<code>sizeof</code> Operator	329
7.9	Pointer Expressions and Pointer Arithmetic	331
7.9.1	Adding Integers to and Subtracting Integers from Pointers	332

7.9.2	Subtracting One Pointer from Another	333
7.9.3	Pointer Assignment	333
7.9.4	Cannot Dereference a <code>void*</code> Pointer	333
7.9.5	Comparing Pointers	334
7.10	Objects-Natural Case Study: C++20 <code>spans</code> —Views of Contiguous Container Elements	334
7.11	A Brief Intro to Pointer-Based Strings	340
7.11.1	Command-Line Arguments	341
7.11.2	Revisiting C++20's <code>to_array</code> Function	342
7.12	Looking Ahead to Other Pointer Topics	344
7.13	Wrap-Up	344

8 `strings`, `string_views`, Text Files, CSV Files and Regex **357**

8.1	Introduction	358
8.2	<code>string</code> Assignment and Concatenation	359
8.3	Comparing <code>strings</code>	361
8.4	Substrings	363
8.5	Swapping <code>strings</code>	364
8.6	<code>string</code> Characteristics	365
8.7	Finding Substrings and Characters in a <code>string</code>	367
8.8	Replacing and Erasing Characters in a <code>string</code>	370
8.9	Inserting Characters into a <code>string</code>	372
8.10	Numeric Conversions	373
8.11	<code>string_view</code>	374
8.12	Files and Streams	377
8.13	Creating a Sequential File	378
8.14	Reading Data from a Sequential File	381
8.15	Reading and Writing Quoted Text	384
8.16	Updating Sequential Files	385
8.17	String Stream Processing	386
8.18	Raw String Literals	389
8.19	Objects-Natural Data Science Case Study: Reading and Analyzing a CSV File Containing <i>Titanic</i> Disaster Data	390
8.19.1	Using <code>rapidcsv</code> to Read the Contents of a CSV File	390
8.19.2	Reading and Analyzing the <i>Titanic</i> Disaster Dataset	392
8.20	Objects-Natural Data Science Case Study: Intro to Regular Expressions	399
8.20.1	Matching Complete Strings to Patterns	400
8.20.2	Replacing Substrings	405
8.20.3	Searching for Matches	405
8.21	Wrap-Up	408

9 Custom Classes **431**

9.1	Introduction	432
-----	--------------	-----

9.2	Test-Driving an Account Object	433
9.3	Account Class with a Data Member and <i>Set</i> and <i>Get</i> Member Functions	435
9.3.1	Class Definition	435
9.3.2	Access Specifiers <code>private</code> and <code>public</code>	437
9.4	Account Class: Custom Constructors	438
9.5	Software Engineering with <i>Set</i> and <i>Get</i> Member Functions	442
9.6	Account Class with a Balance	444
9.7	Time Class Case Study: Separating Interface from Implementation	447
9.7.1	Interface of a Class	448
9.7.2	Separating the Interface from the Implementation	448
9.7.3	Class Definition	449
9.7.4	Member Functions	450
9.7.5	Including the Class Header in the Source-Code File	450
9.7.6	Scope Resolution Operator (<code>::</code>)	451
9.7.7	Member Function <code>setTime</code> and Throwing Exceptions	451
9.7.8	Member Functions <code>to24HourString</code> and <code>to12HourString</code>	451
9.7.9	Implicitly Inlining Member Functions	452
9.7.10	Member Functions vs. Global Functions	452
9.7.11	Using Class <code>Time</code>	452
9.7.12	Object Size	454
9.8	Compilation and Linking Process	454
9.9	Class Scope and Accessing Class Members	455
9.10	Access Functions and Utility Functions	456
9.11	Time Class Case Study: Constructors with Default Arguments	457
9.11.1	Class <code>Time</code>	457
9.11.2	Overloaded Constructors and Delegating Constructors	462
9.12	Destructors	463
9.13	When Constructors and Destructors Are Called	464
9.14	Time Class Case Study: A Subtle Trap — Returning a Reference or a Pointer to a <code>private</code> Data Member	468
9.15	Default Assignment Operator	470
9.16	<code>const</code> Objects and <code>const</code> Member Functions	472
9.17	Composition: Objects as Members of Classes	474
9.18	<code>friend</code> Functions and <code>friend</code> Classes	480
9.19	The <code>this</code> Pointer	482
9.19.1	Implicitly and Explicitly Using the <code>this</code> Pointer to Access an Object's Data Members	482
9.19.2	Using the <code>this</code> Pointer to Enable Cascaded Function Calls	484
9.20	<code>static</code> Class Members: Classwide Data and Member Functions	487
9.21	Aggregates in C++20	492
9.21.1	Initializing an Aggregate	493
9.21.2	C++20 Designated Initializers	493
9.22	Concluding Our Objects Natural Case Study Track: Studying the Vigenère Secret-Key Cipher Implementation	494
9.23	Wrap-Up	507

10 OOP: Inheritance and Runtime Polymorphism 529

10.1	Introduction	530
10.2	Base Classes and Derived Classes	532
10.2.1	CommunityMember Class Hierarchy	533
10.2.2	Shape Class Hierarchy and public Inheritance	534
10.3	Relationship Between Base and Derived Classes	535
10.3.1	Creating and Using a SalariedEmployee Class	535
10.3.2	Creating a SalariedEmployee–SalariedCommissionEmployee Inheritance Hierarchy	538
10.4	Constructors and Destructors in Derived Classes	544
10.5	Intro to Runtime Polymorphism: Polymorphic Video Game	545
10.6	Relationships Among Objects in an Inheritance Hierarchy	546
10.6.1	Invoking Base-Class Functions from Derived-Class Objects	547
10.6.2	Aiming Derived-Class Pointers at Base-Class Objects	549
10.6.3	Derived-Class Member-Function Calls via Base-Class Pointers	550
10.7	Virtual Functions and Virtual Destructors	551
10.7.1	Why <code>virtual</code> Functions Are Useful	551
10.7.2	Declaring <code>virtual</code> Functions	552
10.7.3	Invoking a <code>virtual</code> Function	553
10.7.4	<code>virtual</code> Functions in the SalariedEmployee Hierarchy	553
10.7.5	<code>virtual</code> Destructors	557
10.7.6	<code>final</code> Member Functions and Classes	557
10.8	Abstract Classes and Pure <code>virtual</code> Functions	558
10.8.1	Pure <code>virtual</code> Functions	559
10.8.2	Device Drivers: Polymorphism in Operating Systems	559
10.9	Case Study: Payroll System Using Runtime Polymorphism	560
10.9.1	Creating Abstract Base Class Employee	561
10.9.2	Creating Concrete Derived Class SalariedEmployee	563
10.9.3	Creating Concrete Derived Class CommissionEmployee	565
10.9.4	Demonstrating Runtime Polymorphic Processing	567
10.10	Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”	570
10.11	Program to an Interface, Not an Implementation	573
10.11.1	Rethinking the Employee Hierarchy— <code>CompensationModel</code> Interface	575
10.11.2	Class Employee	575
10.11.3	<code>CompensationModel</code> Implementations	577
10.11.4	Testing the New Hierarchy	579
10.11.5	Dependency Injection Design Benefits	580
10.12	Wrap-Up	581

**11 Operator Overloading,
Copy/Move Semantics and Smart Pointers 587**

11.1	Introduction	588
11.2	Using the Overloaded Operators of Standard Library Class <code>string</code>	590

11.3	Operator Overloading Fundamentals	596
11.3.1	Operator Overloading Is Not Automatic	596
11.3.2	Operators That Cannot Be Overloaded	596
11.3.3	Operators That You Do Not Have to Overload	596
11.3.4	Rules and Restrictions on Operator Overloading	597
11.4	(Downplaying) Dynamic Memory Management with <code>new</code> and <code>delete</code>	598
11.5	Modern C++ Dynamic Memory Management: RAII and Smart Pointers	600
11.5.1	Smart Pointers	601
11.5.2	Demonstrating <code>unique_ptr</code>	601
11.5.3	<code>unique_ptr</code> Ownership	603
11.5.4	<code>unique_ptr</code> to a Built-In Array	603
11.6	MyArray Case Study: Crafting a Valuable Class with Operator Overloading	604
11.6.1	Special Member Functions	605
11.6.2	Using Class <code>MyArray</code>	606
11.6.3	<code>MyArray</code> Class Definition	615
11.6.4	Constructor That Specifies a <code>MyArray</code> 's Size	617
11.6.5	Passing a Braced Initializer to a Constructor	618
11.6.6	Copy Constructor and Copy Assignment Operator	619
11.6.7	Move Constructor and Move Assignment Operator	622
11.6.8	Destructor	626
11.6.9	<code>toString</code> and <code>size</code> Functions	626
11.6.10	Overloading the Equality (<code>==</code>) and Inequality (<code>!=</code>) Operators	627
11.6.11	Overloading the Subscript (<code>[]</code>) Operator	629
11.6.12	Overloading the Unary <code>bool</code> Conversion Operator	630
11.6.13	Overloading the Preincrement Operator	631
11.6.14	Overloading the Postincrement Operator	632
11.6.15	Overloading the Addition Assignment Operator (<code>+=</code>)	633
11.6.16	Overloading the Binary Stream Extraction (<code>>></code>) and Stream Insertion (<code><<</code>) Operators	633
11.6.17	<code>friend</code> Function <code>swap</code>	636
11.7	C++20 Three-Way Comparison Operator (<code><=></code>)	636
11.8	Converting Between Types	640
11.9	<code>explicit</code> Constructors and Conversion Operators	641
11.10	Overloading the Function Call Operator (<code>()</code>)	644
11.11	Wrap-Up	644

12 Exceptions and a Look Forward to Contracts 653

12.1	Introduction	654
12.2	Exception-Handling Flow of Control	658
12.2.1	Defining a Custom Exception Class	658
12.2.2	Demonstrating Exception Handling	659
12.2.3	Enclosing Code in a <code>try</code> Block	660
12.2.4	Defining a <code>catch</code> Handler for <code>DivideByZeroExceptions</code>	661
12.2.5	Termination Model of Exception Handling	662
12.2.6	Flow of Control When the User Enters a Nonzero Denominator	663
12.2.7	Flow of Control When the User Enters a Zero Denominator	663

12.3	Exception Safety Guarantees and <code>noexcept</code>	664
12.4	Rethrowing an Exception	665
12.5	Stack Unwinding and Uncaught Exceptions	667
12.6	When to Use Exception Handling	669
12.6.1	<code>assert</code> Macro	671
12.6.2	Failing Fast	671
12.7	Constructors, Destructors and Exception Handling	672
12.7.1	Throwing Exceptions from Constructors	672
12.7.2	Catching Exceptions in Constructors via Function <code>try</code> Blocks	673
12.7.3	Exceptions and Destructors: Revisiting <code>noexcept(false)</code>	675
12.8	Processing new Failures	676
12.8.1	<code>new</code> Throwing <code>bad_alloc</code> on Failure	677
12.8.2	<code>new</code> Returning <code>nullptr</code> on Failure	678
12.8.3	Handling <code>new</code> Failures Using Function <code>set_new_handler</code>	679
12.9	Standard Library Exception Hierarchy	680
12.10	C++'s Alternative to the <code>finally</code> Block: Resource Acquisition Is Initialization (RAII)	683
12.11	Some Libraries Support Both Exceptions and Error Codes	683
12.12	Logging	685
12.13	Looking Ahead to Contracts	685
12.14	Wrap-Up	694

13 Data Structures: Standard Library Containers and Iterators

697

13.1	Introduction	698
13.2	A Brief Intro to Big <i>O</i>	700
13.3	A Brief Intro to Hash Tables	703
13.4	Introduction to Containers	704
13.4.1	Common Nested Types in Sequence and Associative Containers	706
13.4.2	Common Container Member and Non-Member Functions	707
13.4.3	Requirements for Container Elements	710
13.5	Working with Iterators	710
13.5.1	Using <code>istream_iterator</code> for Input and <code>ostream_iterator</code> for Output	710
13.5.2	Iterator Categories	712
13.5.3	Container Support for Iterators	712
13.5.4	Predefined Iterator Type Names	713
13.5.5	Iterator Operators	713
13.6	A Brief Introduction to Algorithms	714
13.7	Sequence Containers	715
13.8	<code>vector</code> Sequence Container	715
13.8.1	Using <code>vectors</code> and Iterators	716
13.8.2	<code>vector</code> Element-Manipulation Functions	719
13.9	<code>list</code> Sequence Container	723
13.10	<code>deque</code> Sequence Container	728

13.11	Associative Containers	730
13.11.1	<code>multiset</code> Associative Container	730
13.11.2	<code>set</code> Associative Container	734
13.11.3	<code>multimap</code> Associative Container	736
13.11.4	<code>map</code> Associative Container	738
13.12	Container Adaptors	739
13.12.1	<code>stack</code> Adaptor	740
13.12.2	<code>queue</code> Adaptor	742
13.12.3	<code>priority_queue</code> Adaptor	743
13.13	<code>bitset</code> Near Container	744
13.14	Wrap-Up	746

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

773

14.1	Introduction	774
14.2	Algorithm Requirements: C++20 Concepts	776
14.3	Lambdas and Algorithms	778
14.4	Algorithms	781
14.4.1	<code>fill</code> , <code>fill_n</code> , <code>generate</code> and <code>generate_n</code>	781
14.4.2	<code>equal</code> , <code>mismatch</code> and <code>lexicographical_compare</code>	783
14.4.3	<code>remove</code> , <code>remove_if</code> , <code>remove_copy</code> and <code>remove_copy_if</code>	786
14.4.4	<code>replace</code> , <code>replace_if</code> , <code>replace_copy</code> and <code>replace_copy_if</code>	790
14.4.5	Shuffling, Counting, and Minimum and Maximum Element Algorithms	792
14.4.6	Searching and Sorting Algorithms	796
14.4.7	<code>swap</code> , <code>iter_swap</code> and <code>swap_ranges</code>	800
14.4.8	<code>copy_backward</code> , <code>merge</code> , <code>unique</code> , <code>reverse</code> , <code>copy_if</code> and <code>copy_n</code>	802
14.4.9	<code>inplace_merge</code> , <code>unique_copy</code> and <code>reverse_copy</code>	805
14.4.10	Set Operations	807
14.4.11	<code>lower_bound</code> , <code>upper_bound</code> and <code>equal_range</code>	810
14.4.12	<code>min</code> , <code>max</code> and <code>minmax</code>	812
14.4.13	Algorithms <code>gcd</code> , <code>lcm</code> , <code>iota</code> , <code>reduce</code> and <code>partial_sum</code> from Header <code><numeric></code>	813
14.4.14	Heapsort and Priority Queues	816
14.5	Function Objects (Functors)	821
14.6	Projections	825
14.7	C++20 Views and Functional-Style Programming	828
14.7.1	Range Adaptors	828
14.7.2	Working with Range Adaptors and Views	830
14.8	Intro to Parallel Algorithms	834
14.9	Standard Library Algorithm Summary	836
14.10	Future Ranges Enhancements	839
14.11	Wrap-Up	840

15	Templates, C++20 Concepts and Metaprogramming	845
15.1	Introduction	846
15.2	Custom Class Templates and Compile-Time Polymorphism	849
15.3	C++20 Function Template Enhancements	854
15.3.1	C++20 Abbreviated Function Templates	854
15.3.2	C++20 Templated Lambdas	856
15.4	C++20 Concepts: A First Look	856
15.4.1	Unconstrained Function Template <code>multiply</code>	857
15.4.2	Constrained Function Template with a C++20 Concepts <code>requires</code> Clause	860
15.4.3	C++20 Predefined Concepts	862
15.5	Type Traits	864
15.6	C++20 Concepts: A Deeper Look	868
15.6.1	Creating a Custom Concept	868
15.6.2	Using a Concept	869
15.6.3	Using Concepts in Abbreviated Function Templates	870
15.6.4	Concept-Based Overloading	871
15.6.5	<code>requires</code> Expressions	874
15.6.6	C++20 Exposition-Only Concepts	877
15.6.7	Techniques Before C++20 Concepts: SFINAE and Tag Dispatch	878
15.7	Testing C++20 Concepts with <code>static_assert</code>	879
15.8	Creating a Custom Algorithm	881
15.9	Creating a Custom Container and Iterators	883
15.9.1	Class Template <code>ConstIterator</code>	885
15.9.2	Class Template <code>Iterator</code>	888
15.9.3	Class Template <code>MyArray</code>	890
15.9.4	<code>MyArray</code> Deduction Guide for Braced Initialization	893
15.9.5	Using <code>MyArray</code> with <code>std::ranges</code> Algorithms	894
15.10	Default Arguments for Template Type Parameters	898
15.11	Variable Templates	898
15.12	Variadic Templates and Fold Expressions	899
15.12.1	<code>tuple</code> Variadic Class Template	899
15.12.2	Variadic Function Templates and an Intro to Fold Expressions	902
15.12.3	Types of Fold Expressions	906
15.12.4	How Unary Fold Expressions Apply Their Operators	906
15.12.5	How Binary-Fold Expressions Apply Their Operators	909
15.12.6	Using the Comma Operator to Repeatedly Perform an Operation	910
15.12.7	Constraining Parameter Pack Elements to the Same Type	911
15.13	Template Metaprogramming	913
15.13.1	C++ Templates Are Turing Complete	914
15.13.2	Computing Values at Compile-Time	914
15.13.3	Conditional Compilation with Template Metaprogramming and <code>constexpr if</code>	919
15.13.4	Type Metafunctions	921
15.14	Wrap-Up	925

16 C++20 Modules: Large-Scale Development 933

16.1	Introduction	934
16.2	Compilation and Linking Before C++20	936
16.3	Advantages and Goals of Modules	937
16.4	Example: Transitioning to Modules—Header Units	938
16.5	Modules Can Reduce Translation Unit Sizes and Compilation Times	941
16.6	Example: Creating and Using a Module	942
16.6.1	module Declaration for a Module Interface Unit	943
16.6.2	Exporting a Declaration	945
16.6.3	Exporting a Group of Declarations	945
16.6.4	Exporting a namespace	945
16.6.5	Exporting a namespace Member	946
16.6.6	Importing a Module to Use Its Exported Declarations	946
16.6.7	Example: Attempting to Access Non-Exported Module Contents	948
16.7	Global Module Fragment	951
16.8	Separating Interface from Implementation	951
16.8.1	Example: Module Implementation Units	951
16.8.2	Example: Modularizing a Class	954
16.8.3	:private Module Fragment	958
16.9	Partitions	958
16.9.1	Example: Module Interface Partition Units	959
16.9.2	Module Implementation Partition Units	962
16.9.3	Example: “Submodules” vs. Partitions	962
16.10	Additional Modules Examples	967
16.10.1	Example: Importing the C++ Standard Library as Modules	967
16.10.2	Example: Cyclic Dependencies Are Not Allowed	969
16.10.3	Example: imports Are Not Transitive	970
16.10.4	Example: Visibility vs. Reachability	971
16.11	Migrating Code to Modules	972
16.12	Future of Modules and Modules Tooling	973
16.13	Wrap-Up	975

17 Parallel Algorithms and Concurrency: A High-Level View 987

17.1	Introduction	988
17.2	Standard Library Parallel Algorithms	991
17.2.1	Example: Profiling Sequential and Parallel Sorting Algorithms	991
17.2.2	When to Use Parallel Algorithms	994
17.2.3	Execution Policies	995
17.2.4	Example: Profiling Parallel and Vectorized Operations	996
17.2.5	Additional Parallel Algorithm Notes	998
17.3	Multithreaded Programming	999
17.3.1	Thread States and the Thread Life Cycle	999
17.3.2	Deadlock and Indefinite Postponement	1001

17.4	Launching Tasks with <code>std::jthread</code>	1003
17.4.1	Defining a Task to Perform in a Thread	1003
17.4.2	Executing a Task in a <code>jthread</code>	1005
17.4.3	How <code>jthread</code> Fixes <code>thread</code>	1007
17.5	Producer–Consumer Relationship: A First Attempt	1008
17.6	Producer–Consumer: Synchronizing Access to Shared Mutable Data	1015
17.6.1	Class <code>SynchronizedBuffer</code> : Mutexes, Locks and Condition Variables	1017
17.6.2	Testing <code>SynchronizedBuffer</code>	1023
17.7	Producer–Consumer: Minimizing Waits with a Circular Buffer	1027
17.8	Readers and Writers	1036
17.9	Cooperatively Canceling <code>jthreads</code>	1037
17.10	Launching Tasks with <code>std::async</code>	1040
17.11	Thread-Safe, One-Time Initialization	1047
17.12	A Brief Introduction to Atomics	1048
17.13	Coordinating Threads with C++20 Latches and Barriers	1052
17.13.1	C++20 <code>std::latch</code>	1052
17.13.2	C++20 <code>std::barrier</code>	1055
17.14	C++20 Semaphores	1058
17.15	C++23: A Look to the Future of C++ Concurrency	1062
17.15.1	Parallel Ranges Algorithms	1062
17.15.2	Concurrent Containers	1062
17.15.3	Other Concurrency-Related Proposals	1063
17.16	Wrap-Up	1063

18 C++20 Coroutines

1073

18.1	Introduction	1074
18.2	Coroutine Support Libraries	1075
18.3	Installing the <code>concurrency</code> and <code>generator</code> Libraries	1077
18.4	Creating a Generator Coroutine with <code>co_yield</code> and the <code>generator</code> Library	1077
18.5	Launching Tasks with <code>concurrency</code>	1081
18.6	Creating a Coroutine with <code>co_await</code> and <code>co_return</code>	1085
18.7	Low-Level Coroutines Concepts	1093
18.8	Future Coroutines Enhancements	1096
18.9	Wrap-Up	1096

19 Stream I/O & C++20 Text Formatting

1101

19.1	Introduction	1102
19.2	Streams	1102
19.2.1	Classic Streams vs. Standard Streams	1103
19.2.2	<code>iostream</code> Library Headers	1103
19.2.3	Stream Input/Output Classes and Objects	1103

19.3	Stream Output	1104
19.3.1	Output of <code>char*</code> Variables	1105
19.3.2	Character Output Using Member Function <code>put</code>	1106
19.4	Stream Input	1106
19.4.1	<code>get</code> and <code>getline</code> Member Functions	1106
19.4.2	<code>istream</code> Member Functions <code>peek</code> , <code>putback</code> and <code>ignore</code>	1109
19.5	Unformatted I/O Using <code>read</code> , <code>write</code> and <code>gcount</code>	1110
19.6	Stream Manipulators	1111
19.6.1	Integral Stream Base: <code>dec</code> , <code>oct</code> , <code>hex</code> and <code>setbase</code>	1112
19.6.2	Floating-Point Precision (<code>setprecision</code> , <code>precision</code>)	1112
19.6.3	Field Width (<code>width</code> , <code>setw</code>)	1114
19.6.4	User-Defined Output Stream Manipulators	1115
19.6.5	Trailing Zeros and Decimal Points (<code>showpoint</code>)	1116
19.6.6	Alignment (<code>left</code> , <code>right</code> and <code>internal</code>)	1117
19.6.7	Padding (<code>fill</code> , <code>setfill</code>)	1118
19.6.8	Integral Stream Base (<code>dec</code> , <code>oct</code> , <code>hex</code> , <code>showbase</code>)	1119
19.6.9	Floating-Point Numbers; Scientific and Fixed Notation (<code>scientific</code> , <code>fixed</code>)	1120
19.6.10	Uppercase/Lowercase Control (<code>uppercase</code>)	1121
19.6.11	Specifying Boolean Format (<code>boolalpha</code>)	1122
19.6.12	Setting and Resetting the Format State via Member Function <code>flags</code>	1123
19.7	Stream Error States	1124
19.8	Tying an Output Stream to an Input Stream	1127
19.9	C++20 Text Formatting	1127
19.9.1	C++20 <code>std::format</code> Presentation Types	1128
19.9.2	C++20 <code>std::format</code> Field Widths and Alignment	1130
19.9.3	C++20 <code>std::format</code> Numeric Formatting	1131
19.9.4	C++20 <code>std::format</code> Field Width and Precision Placeholders	1132
19.10	Wrap-Up	1133

20 Other Topics and a Look Toward the Future of C++

1141

20.1	Introduction	1142
20.2	<code>shared_ptr</code> and <code>weak_ptr</code> Smart Pointers	1143
20.2.1	Reference Counted <code>shared_ptr</code>	1143
20.2.2	<code>weak_ptr</code> : <code>shared_ptr</code> Observer	1147
20.3	Runtime Polymorphism with <code>std::variant</code> and <code>std::visit</code>	1154
20.4	<code>protected</code> Class Members: A Deeper Look	1160
20.5	Non-Virtual Interface (NVI) Idiom	1161
20.6	Inheriting Base-Class Constructors	1168
20.7	Multiple Inheritance	1169
20.7.1	Diamond Inheritance	1174
20.7.2	Eliminating Duplicate Subobjects with <code>virtual</code> Base-Class Inheritance	1176

20.8	<code>public</code> , <code>protected</code> and <code>private</code> Inheritance	1177
20.9	namespaces: A Deeper Look	1179
20.9.1	Defining namespaces	1180
20.9.2	Accessing namespace Members with Qualified Names	1181
20.9.3	using Directives Should Not Be Placed in Headers	1181
20.9.4	Nested Namespaces	1181
20.9.5	Aliases for namespace Names	1181
20.10	Storage Classes and Storage Duration	1182
20.10.1	Storage Duration	1182
20.10.2	Local Variables and Automatic Storage Duration	1182
20.10.3	Static Storage Duration	1183
20.10.4	<code>mutable</code> Class Members	1184
20.11	Operator Keywords	1185
20.12	<code>decltype</code> Operator	1186
20.13	Trailing Return Types for Functions	1187
20.14	<code>[[nodiscard]]</code> Attribute	1187
20.15	Some Key C++23 Features	1189
20.16	Wrap-Up	1194

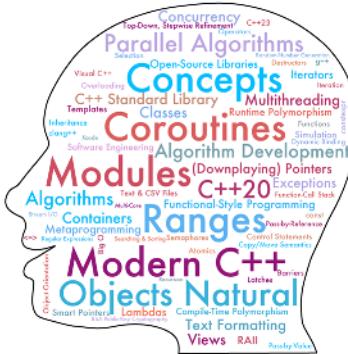
21 Computer Science Thinking: Searching, Sorting and Big O

1197

21.1	Introduction	1198
21.2	Efficiency of Algorithms: Big O	1199
21.2.1	$O(1)$ Algorithms	1199
21.2.2	$O(n)$ Algorithms	1199
21.2.3	$O(n^2)$ Algorithms	1200
21.3	Linear Search	1201
21.3.1	Implementation	1201
21.3.2	Efficiency of Linear Search	1202
21.4	Binary Search	1203
21.4.1	Implementation	1203
21.4.2	Efficiency of Binary Search	1207
21.5	Insertion Sort	1208
21.5.1	Implementation	1209
21.5.2	Efficiency of Insertion Sort	1210
21.6	Selection Sort	1210
21.6.1	Implementation	1211
21.6.2	Efficiency of Selection Sort	1213
21.7	Merge Sort (A Recursive Implementation)	1213
21.7.1	Implementation	1214
21.7.2	Efficiency of Merge Sort	1219
21.7.3	Summarizing Various Algorithms' Big O Notations	1219
21.8	Wrap-Up	1221

Index

1225



Preface

I An Innovative Modern C++ Programming Textbook

Good programmers write code that humans can understand.¹

—Martin Fowler

Welcome to *C++ How to Program: An Objects-Natural Approach, 11/e*. We present a friendly, contemporary, code-intensive, case-study-oriented introduction to C++, the world's third most popular programming language according to the TIOBE Index.²

C++ is popular for building high-performance business-critical and mission-critical computing systems—operating systems, real-time systems, embedded systems, game systems, banking systems, air-traffic-control systems, communications systems and more. This book is an introductory- through intermediate-level college textbook presentation of the C++20 version of C++ and its associated standard libraries, with a look toward C++23 and C++26. In this Preface, we present the “soul of the book.”

Live-Code Approach and Getting the Code

At the heart of the book is the Deitel signature **live-code approach**. Rather than code snippets, we show C++ as it's intended to be used in the context of 255 complete, working, real-world C++ programs with live outputs.

Read the Before You Begin section that follows this Preface to learn how to set up your Windows, macOS or Linux computer to run the code examples. For your convenience, we provide the book's examples in C++ source-code (.cpp and .h) files for use with integrated development environments and command-line compilers. All the source code is available for download at

- <https://github.com/pdeitel/CPlusPlusHowToProgram11e>
- <https://www.deitel.com/cpphtp11>

Chapter 1's Test-Drives (Section 1.11) shows how to compile and run the code examples with each of our preferred compilers. Executing each program in parallel with reading the text will make your learning experience “come alive.” If you encounter a problem, you can reach us at deitel@deitel.com, and we'll respond promptly.

1. Martin Fowler (with contributions by Kent Beck). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. 2018.

2. Tiobe Index for February 2023. Accessed March 8, 2023. <https://www.tiobe.com/tiobe-index/>.

Key Computing Trends

For many decades:

- computer hardware has rapidly been getting faster, cheaper and smaller,
- Internet bandwidth (that is, its information-carrying capacity) has rapidly been getting larger and cheaper, and
- quality computer software has become ever more abundant and often free or nearly free through the open-source movement.

The Internet of Things (IoT) already connects tens of billions of computerized devices of every imaginable type, and that number is likely to grow quickly. These generate enormous volumes of data (one form of “big data”) at rapidly increasing speeds and quantities. And most computing will eventually be performed in “the Cloud”—that is, by using computing services accessible over the Internet.

For the novice, the book’s early chapters establish a solid foundation in programming fundamentals. The mid-range to high-end chapters and the 50 more significant case study examples and case study exercises will ease you into professional software-development challenges and practices.

Given the extraordinary performance demands that today’s applications place on computer hardware, software and the Internet, professionals often choose C++ to build the most performance-intensive portions of these applications. Throughout the book, we emphasize performance issues to help you prepare for industry.

2 Modern C++

We cover Modern C++—C++20, C++17, C++14 and C++11—with a look toward key features coming in C++23 and anticipated for C++26. We employ industry best practices, emphasizing Modern C++ idioms—which change how developers write C++ programs—while focusing on performance, security and software engineering. We present rich treatments of C++20’s “big four” features—ranges, concepts, modules and coroutines. We’ll say more about these in Section 6 of this Preface.

3 Target Audiences

The book’s modular architecture (see the diagram on the next page) makes it appropriate for several audiences:

- Introductory and intermediate college programming courses in Computer Science, Computer Engineering, Information Systems, Information Technology, Software Engineering and related disciplines.
- Science, technology, engineering and math (STEM) college courses with a programming component.
- Professional industry training courses.
- Experienced professionals learning the latest Modern C++ idioms to prepare for upcoming projects.

C++ How to Program: An Objects-Natural Approach, 11/e

by Paul Deitel & Harvey Deitel

PART 1 C++20 Fundamentals Quickstart & Procedural Programming

PART 2 Containers, C++20 Ranges, Pointers, Strings & Files

1. Intro: Test-Driving Popular, Free C++ Compilers
Intro to Hardware, Software & Internet; Test-Driving the Visual C++, GNU g++ and LLVM clang++ compilers.

2. Intro to C++20 Programming
C++ fundamentals. “Objects-Natural” (ON) approach intro—using libraries to build powerful object-oriented applications with few lines of code.

3. Control Statements, Part 1
Intro to C++20 text formatting.
ON: Super-Sized Integers with the Boost Multiprecision Library

4. Control Statements, Part 2
ON: Precise Monetary Calculations with the Boost Multiprecision Library

5. Functions and an Intro to Function Templates

ON: InfyIun Lhqtomh Wjtz Qarv:
Qjwazkrpm xZndmwwqhlz
(encrypted title for our private-key cryptography case study)

PART 3 Modern Object-Oriented Programming & Exceptions

6. arrays, vectors, Ranges and Functional-Style Programming

Intro to functional-style programming.
ON: Class Template vector

7. (Down)playing Pointers in Modern C++

Security & safe programming.
ON: C++20 spans

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

ON: Reading and Analyzing the Titanic Disaster Data (CSV)
ON: Intro to Regular Expressions



HOW TO PROGRAM

An Objects-Natural Approach



PAUL DEITEL
HARVEY DEITEL
ELEVENTH EDITION

PART 5, Advanced Topics: Modules, Parallel Algorithms, Concurrency & Coroutines

PART 6 C++20 Modules: Large-Scale Development

Import, header units, module declarations, module fragments, partitions

PART 7 Parallel Algorithms & Concurrency: A High-Level View

Multi-core performance with C++17 parallel algorithms, concurrency, multithreading

PART 8 C++20 Coroutines

co_await, co_return,

coroutines support libraries, generators, executors and tasks

PART 9 Miscellaneous Topics

10. Custom Classes

ON: Studying the Vigenère Secret-Key Cipher implementation

11. OOP: Inheritance and Runtime Polymorphism

Programming to an interface.

12. Operator Overloading, Copy/Move Semantics, Smart Pointers and RAII

Crafting valuable classes: Custom MyArray class, C++20 three-way comparison operator <=>, resource management via RAII (Resource Acquisition Is Initialization)

13. Standard Library Containers and Iterators

Manipulating standard data structures

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

Functional-style programming

15. Templates, C++20 Concepts and Metaprogramming

Compile-time polymorphism, function templates, C++20 abbreviated function templates, class templates, variadic templates, fold expressions

PART 19 Stream I/O and C++20 Text Formatting

20. Other Topics and a Look Toward C++23 and C++26

21. Computer Science Thinking: Searching, Sorting and Big O

Programming tips: C++ Core Guidelines, Software Engineering, Performance, Security, Errors, C++20 Modules, C++20 Concepts, Data Science.

g++ & clang++ Docker containers.

A look toward C++23 and C++26.

Blog: <https://deitel.com/blog>.

- Static code-analysis tools.
- Use developer resources: GitHub®, StackOverflow®, open-source, more.

<https://deitel.com/cphptp11>.

4 “Objects-Natural” Learning Approach

Traditionally object-oriented programming textbooks have been designated as “late objects” or “early objects.” What’s really “late” or “early” in these textbooks is not “objects.” Rather, it’s teaching how to develop custom classes—the blueprints from which objects are built. We’ve written textbooks using both of these approaches.

What Is “Objects Natural?”

As we wrote our Python textbook,³ we noticed that although our presentation fit the “late objects” model, it was actually something more—and *that* something is special. We call it the “objects-natural approach,” and we’re now applying it to C++ and the other object-oriented programming languages we write about.

Similar to “late objects,” our objects-natural approach begins with programming fundamentals, but you’ll work extensively in the early chapters with easy-to-use powerful pre-existing classes that do significant things. You’ll quickly create objects of those classes (typically with one line of code) and tell them to “strut their stuff” with a minimal number of simple C++ statements.

Even if you’re a programming novice, you can perform significant tasks long before you learn how to create custom C++ classes in Chapter 9. This is one of the most compelling aspects of working with a mature object-oriented language like C++. After covering programming fundamentals with the objects-natural approach, we provide a deep treatment of object-oriented programming beginning with a rich treatment of custom class creation.

An Abundance of Free Classes

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

- the C++ standard library,
- platform-specific libraries, such as those provided with Microsoft Windows, Apple macOS or various Linux versions, and
- free third-party C++ libraries, often created by the open-source community.

We encourage you to view lots of free, open-source C++ code available on sites like GitHub. Reading other programmers’ code is a great way to learn.

The Boost Project

Boost provides 168 powerful open-source C++ libraries⁴ and serves as a “breeding ground” for new capabilities that might eventually be incorporated into the C++ standard libraries. The following StackOverflow post lists Modern C++ libraries and language features that evolved from the Boost libraries:⁵

<https://stackoverflow.com/a/8852421>

-
3. *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud* (<https://deitel.com/pycds>).
 4. “Boost 1.81.0 Library Documentation.” Accessed March 8, 2023. https://www.boost.org/doc/libs/1_81_0/.
 5. Kennytm, Answer to “Which Boost Features Overlap with C++11?” Accessed March 8, 2023. <https://stackoverflow.com/a/8852421>.

We use the Boost Multiprecision library in our **objects-natural case studies** on **super-sized integers** (Section 3.14) and **precise monetary calculations** (Section 4.14).

Objects-Natural Case Studies

Chapter 1 presents a friendly introduction to the basic concepts and terminology of object technology. In the early chapters, you'll create and use objects of preexisting classes long before Chapter 9 discusses how to create custom classes. See Section 6's Tour of the Book for descriptions of our objects-natural case studies in Chapters 2–9. A perfect example of the objects-natural approach is using objects of standard library classes, like `array` and `vector` (Chapter 6), without knowing how to write classes in general or how those classes are implemented in particular. Throughout the rest of the book, we use C++ standard library capabilities extensively.

5 Programming Wisdom and Key C++20 Features

We integrate smoothly into the flow of the text software-development wisdom, data science topics, C++20 modules and C++20 concepts features:

- Software engineering observations highlight architectural and design issues for proper software construction, especially for larger systems.  SE
- Security best practices help you strengthen your programs against attacks.  Sec
- Performance tips highlight opportunities to make your programs run faster or minimize the amount of memory they occupy.  Perf
- Common programming errors help reduce the likelihood that you'll make the same mistakes.  Err
- C++ Core Guidelines recommendations (introduced in Section 12).  CG
- C++20's new modules features.  Mod
- C++20's new concepts features.  Concepts
- We present and use data science topics in several examples and exercises.  DS

6 Tour of the Book

The one-page Table of Contents diagram earlier in this Preface provides a high-level overview of the book's modular architecture from “40,000 feet.” We recommend that you refer to that diagram as you read this section.

The early chapters establish a solid foundation in C++20 fundamentals. The mid-range to high-end chapters introduce Modern C++ software development. We discuss C++'s programming models:

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

Whether you’re a student getting a sense of the textbook you’ll be using, an instructor planning your course syllabus or a professional software developer deciding which chapters to read as you prepare for a project, this detailed Tour of the Book will help you make the best decisions.

Part I: Programming Fundamentals Quickstart

Chapter 1, Intro and Test-Driving Popular, Free C++ Compilers, engages programming novices with intriguing facts and figures to excite them about studying computers and computer programming. The chapter includes current technology trends, hardware, software and Internet concepts, and a sample data hierarchy from bits to bytes, fields, records and databases. It lays the groundwork for the C++ programming discussions in Chapters 2–21 and the substantial integrated case study examples, exercises and projects.

We discuss the programming-language types and technologies you’ll likely use as you develop software. We introduce the C++ standard library—existing, reusable, top-quality, high-performance capabilities that help you avoid “reinventing the wheel.” You’ll enhance your productivity by using libraries to perform significant tasks while writing only modest numbers of instructions. We also introduce the Internet, the World Wide Web, the “Cloud” and the Internet of Things (IoT), laying the groundwork for modern applications development.

This chapter’s test-drives demonstrate how to compile and execute C++ code with three of the most popular C++ development environments:

- Microsoft’s Visual C++ in Visual Studio on Windows,
- GNU’s `g++` on macOS/Linux and
- The LLVM Compiler Infrastructure’s `clang++` on macOS/Linux.

We tested the book’s 255 code examples using each compiler.⁶ Choose whichever you prefer—the book also works well with many others. See the Before You Begin section that follows this Preface for compiler installation instructions.

We also demonstrate running `g++` and `clang++` using Docker containers. Docker is an important tool that enables you to run the latest versions of these compilers on Windows, macOS or Linux. See Section 7 of this Preface for more details on Docker and Docker containers.

You’ll learn just how big “big data” is and how quickly it’s getting even bigger. The chapter closes with an introduction to artificial intelligence (AI)—a key overlap between computer-science and data-science. AI and data science will likely play significant roles in your computing career.

Chapter 2, Intro to C++ Programming, presents C++ fundamentals and illustrates key language features, including input, output, fundamental data types, arithmetic operators and their precedence, and decision-making. As part of our objects-natural approach, Section 2.8’s **objects-natural case study** demonstrates **creating and using objects of the C++ standard library class `string`**—without you having to know how to develop custom classes in general or how the large complex class `string` is implemented in particular).

6. We point out the few cases in which a compiler does not support a particular feature.

Chapter 3, Algorithm Development and Control Statements: Part 1, is one of the most important chapters for programming novices. It focuses on problem-solving and algorithm development with C++’s control statements. You’ll develop algorithms through top-down, stepwise refinement, using the `if` and `if...else` selection statements, the `while` iteration statement for counter-controlled and sentinel-controlled iteration, and the increment, decrement and assignment operators. The chapter presents three **algorithm-development case studies**—Counter-Controlled Iteration, Sentinel-Controlled Iteration and Nested Control Statements. Section 3.14’s **objects-natural case study** demonstrates **using the open-source Boost Multiprecision library’s `cpp_int` class to create super-sized integers.**

Chapter 4, Control Statements: Part 2, presents C++’s other control statements—for, `do...while`, `switch`, `break` and `continue`—and the logical operators. A key feature of this chapter is its **structured-programming summary**. We introduce C++20’s `format` function, which provides powerful new text-formatting capabilities. Pre-C++20 text formatting is complex and verbose. The `format` function greatly simplifies data formatting using a concise syntax based on the Python programming language’s text formatting. We present a few C++20 text-formatting features throughout the book, then take a deeper look in Chapter 19. In introductory computer science courses, instructors may wish to present Section 19.9 after C++20 text formatting is introduced in Chapter 4. Section 4.14’s **objects-natural case study** demonstrates **using the open-source Boost Multiprecision library’s `cpp_dec_float_50` class for precise monetary calculations.**

Chapter 5, Functions and an Intro to Function Templates, introduces custom functions. We introduce random-number generation and simulation techniques and use them in our first of several **Random-Number Simulation case studies** throughout the book to **implement a popular casino dice game**. We discuss C++’s secure library of random-number capabilities that can produce “nondeterministic” random numbers—a set of random numbers that can’t be predicted. Such random-number generators are used in simulations and security scenarios where predictability is undesirable. We also discuss passing information between functions and how the function-call stack and stack frames support the function call/return mechanism. We begin our rich treatment of the powerful computer science topic of recursion.



Section 5.19’s **objects-natural case study** title—**Pqyoaf X Nylfomigrob Qwbbfmh Mndogyk: Rboqlrut yua Boklnxhmwyex**—looks like gibberish. This is not a mistake! This case study continues our security emphasis by introducing **cryptography**, which 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. You’ll use our implementation of the Vigenère secret-key cipher⁷ algorithm to encrypt and decrypt messages and to decrypt this section’s title. Then, in Chapter 9’s **objects-natural case study**, you’ll study our class that implements the **Vigenère secret-key cipher** using classes and array-processing techniques. In **Chapter 9 case study exercises**, you’ll also explore far more secure **public-key cryptography** with the RSA algorithm.

7. “Vigenère Cipher.” Accessed April 29, 2023. https://en.wikipedia.org/wiki/Vigenère_cipher.

Part 2: Arrays, Pointers and Strings

Chapter 6, arrays, vectors, Ranges and Functional-Style Programming, begins our early coverage of the C++ standard library’s containers, iterators and algorithms. We present the C++ standard library’s array container for representing lists and tables of values. You’ll define and initialize arrays and access their elements. We discuss passing arrays to functions, sorting and searching arrays and manipulating multidimensional arrays.

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 is easier to read, debug and modify. Chapter 6 begins our introduction to functional-style programming using arrays, lambda expressions (anonymous functions), and C++20 ranges—a C++20 “big four” feature that simplifies how you call many of the C++ standard library’s predefined algorithms. We continue that discussion in Chapters 13 and 14.

Section 6.15’s **objects-natural case study** demonstrates the C++ standard library class **template vector**. Chapter 6 is essentially a large **objects-natural case study of both arrays and vectors**. The code in this chapter is a good example of Modern C++ coding idioms. This chapter’s exercises include a case study on the famous **Knight’s Tour problem**, which we approach in various ways, including an AI strategy called **heuristic programming**.

Chapter 7, (Downplaying) Pointers in Modern C++, explains pointer concepts, such as declaring pointers, initializing pointers, getting the memory address of a variable, dereferencing pointers, pointer arithmetic and the close relationship among built-in pointers, pointer-based arrays and pointer-based strings, each of which C++ inherited from the C programming language. Pointers are powerful but challenging to work with. So, we focus on Modern C++ features that eliminate the need for most pointers and make your code more robust and secure, including references, “smart pointer” objects, arrays and vectors, strings, C++20 spans and C++17 string_views. We still cover built-in arrays because they remain useful in C++, and so you’ll be able to read legacy C++ code that you’ll encounter in industry. In new development, you should favor Modern C++ capabilities. Section 7.10’s **objects-natural case study** demonstrates one such capability—**C++20 spans**. These enable you to view and manipulate elements of contiguous containers, such as pointer-based arrays and standard library arrays and vectors, without using pointers directly.

In a **Random-Number Simulation case study exercise**, you’ll implement the famous race between the tortoise and the hare. This chapter also contains the first of our two **systems programming case study exercises**—**Building Your Own Computer** (as a virtual machine). In the context of several **case study exercises**, you’ll “peel open” a hypothetical computer and look at its internal structure. We introduce simple machine-language programming and write several small machine-language programs for this computer, which we call the Simpletron. As its name implies, it’s a simple machine, but as you’ll see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands—that is, Simpletron Machine Language, or SML for short. To make this an especially valuable experience, you’ll then build a computer (through the technique of software-based simulation) on which you can actually run your machine-language programs! The Simpletron experience will give you a basic introduction to **virtual machines**—one of the most important systems-architecture concepts in modern computing. Chapter 13 contains the intimately related **systems programming case study exercise**—**Building Your Own Compiler**.



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

- Section 8.19 introduces data analytics by reading and analyzing a CSV file containing the **Titanic Disaster** dataset.
- Section 8.20 introduces **regular-expression pattern matching and text replacement**.

This chapter includes three AI/Data Science case study exercises. In the first case study, **Machine Learning with Simple Linear Regression: Statistics Can Be Deceiving**, you'll learn that an essential aspect of data analytics is “getting to know your data.” One way to do this is via descriptive statistics—but these can be deceiving. To illustrate this, we'll consider visualizations of **Anscombe's Quartet**⁸ (Exercise 8.40)—a famous example of four dramatically different datasets containing x - y coordinate pairs with nearly identical descriptive statistics. You'll then study a **completely coded example** in which we use the popular AI/machine-learning statistical technique called **simple linear regression** that, given a collection of x - y coordinate pairs representing an **independent variable** (x) and a **dependent variable** (y), determines the equation of a straight line ($y = mx + b$) that most closely fits the data. This equation describes the relationship between the dependent and independent variables, enabling us to predict y 's value for any given x . It also allows us to plot a **regression line**. You'll see that the **regression lines for Anscombe's Quartet** are visually identical for all four dramatically different datasets. The code you'll study uses the popular **open-source gnuplot package** to create attractive **visualizations** of Anscombe's Quartet. The gnuplot package uses its own plotting language, different from C++, so we provide extensive code comments that explain the gnuplot commands.

In the AI/Data Science case study exercise, **Machine Learning with Simple Linear Regression: Time Series Analysis** (Exercise 8.41), you'll use what you learned in the preceding exercise to analyze a **time series**, a sequence of values (called **observations**) associated with points in time. Time series examples include daily closing stock prices, hourly temperature readings, the changing positions of a plane in flight, annual crop yields and quarterly company profits. You'll run a **simple linear regression** on 126 years of New York City **average January temperature data** (stored in a CSV file) and use **gnuplot** to plot the data and the regression line so you can determine if there is a cooling or warming trend.

This chapter's final AI/Data Science case study (Exercise 8.42) presents an intro to **similarity detection with very basic natural language processing (NLP)**—an important data science and artificial intelligence topic. NLP helps computers understand, analyze and process text. While writing this book, we used the paid (NLP) tool Grammarly⁹ to help tune the writing and ensure the text's readability for a broad audience. Some people believe

8. “Anscombe's quartet.” Accessed March 8, 2023. https://en.wikipedia.org/wiki/Anscombe%27s_quartet.

that the works of William Shakespeare actually might have been penned by Christopher Marlowe or Sir Francis Bacon, among others.^{10,11} In this exercise, you'll use array-, string- and file-processing techniques to perform simple **similarity detection** on Shakespeare's *Romeo and Juliet* and Marlowe's *Edward the Second*. You'll determine how alike they are by comparing the statistics you calculate, such as the percentages of each unique word among all words in each play. You may be surprised by the results.

Part 3: Object-Oriented Programming



Chapter 9, Custom Classes, begins our substantially upgraded, multi-chapter, Modern C++, object-oriented programming treatment. C++ is extensible—each class you create becomes a new type you can use to create objects. In Chapters 9–11, you'll learn C++'s features for crafting valuable classes and manipulating objects of these classes.

In Section 9.22, we conclude our **objects natural case study track** by studying the Vigenère Secret-Key Cipher class implementation that we demonstrated in Chapter 5's objects-natural case study. Objects-natural case study Exercises 9.32–9.33 demonstrate how to **serialize objects** with JSON (JavaScript Object Notation)—a popular human-and-machine-readable data format commonly used to transmit data over the Internet.

In the context of several **Random-Number Simulation case study exercises**, you'll use arrays of strings, random-number generation and simulation techniques to implement a text-based, card-shuffling-and-dealing program.



In a **Security and Cryptography case study exercise**, you'll also explore public-key cryptography with the RSA algorithm. This technique performs encryption with a public key known to every sender who might want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. Messages can be decrypted only with a paired private key known only to the receiver, so it's much more secure than secret keys in secret-key cryptography. RSA is among the world's most widely used public-key cryptography technologies. You'll build a working, small-scale, classroom version of the RSA cryptosystem.

Chapter 10, OOP: Inheritance and Runtime Polymorphism, focuses on the relationships among classes in an inheritance hierarchy and the powerful runtime polymorphic processing capabilities (for “programming in the general”) that these relationships enable. In this chapter's **runtime-polymorphism case study**, you'll implement an **Employee** class hierarchy in an application that performs polymorphic payroll calculations.

An important aspect of this chapter is understanding how polymorphism works. A key feature of the chapter is its detailed diagram and explanation of how C++ typically implements polymorphism, **virtual** functions and dynamic binding “under the hood.” You'll see that it can use an elegant pointer-based data structure. We also discuss programming to an interface, not an implementation. In Chapter 20, we discuss other more advanced mechanisms for achieving runtime polymorphism, including the non-virtual interface idiom (NVI) and `std::variant/std::visit`.

-
9. Grammarly has free and paid versions (<https://www.grammarly.com>). They provide free plug-ins you can use in several popular web browsers.
 10. “Did Shakespeare Really Write His Own Plays?” Accessed November 13, 2020. <https://www.history.com/news/did-shakespeare-really-write-his-own-plays>.
 11. “Shakespeare authorship question.” Accessed November 13, 2020. https://en.wikipedia.org/wiki/Shakespeare_authorship_question.

Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, shows how to enable C++’s existing operators to work with custom class objects and introduces smart pointers and dynamic memory management. Smart pointers help you avoid dynamic memory management errors and “resource leaks” by providing additional functionality beyond that of built-in pointers. We discuss `unique_ptr` in this chapter and `shared_ptr` and `weak_ptr` in Chapter 20, Other Topics and a Look Toward the Future of C++.

 Err

A key aspect of Chapter 11 is crafting valuable classes. We begin with a `string` class test-drive, presenting an elegant use of operator overloading before you implement your own customized class with overloaded operators. Then, in our **Crafting Valuable Classes case study—one of the book’s most important examples**—you’ll build your own custom `MyArray` class using overloaded operators and other capabilities to solve various problems with C++’s native pointer-based arrays.¹² We introduce and implement the five special member functions you can define in each class—the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. We discuss copy semantics and move semantics, which enable a compiler to move resources from one object to another to avoid costly, unnecessary copies. We introduce C++20’s three-way comparison operator (`<=>`; also called the “spaceship operator”) and show how to implement custom conversion operators. In Chapter 15, you’ll convert a portion of the `MyArray` class into a class template that can store elements of a specified type. You will then have truly “crafted valuable classes.”

 Perf

Chapter 12, Exceptions and a Look Forward to Contracts, continues our exception-handling discussion that began in Chapter 6. We’ve enhanced Chapter 12’s coverage with discussions of when to use exceptions, exception safety guarantees, and using exceptions in the context of constructors and destructors. We show how to handle dynamic memory allocation failures. We 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. The chapter concludes with a case study that introduces contracts—a possible C++26 feature. One goal of contracts is to make most functions `noexcept`—meaning they do not throw exceptions—which might enable the compiler to perform additional optimizations and eliminate the overhead and complexity of exception handling. Another goal is to find errors faster, eliminate them during development and, hopefully, create more robust code for deployment. We introduce preconditions, post-conditions and assertions, and we discuss how they can be implemented as contracts that are tested at execution time. We demonstrate the example code using GCC’s experimental contracts implementation on <https://godbolt.org>.

 Err Perf

Part 4: Standard Library Containers, Iterators and Algorithms

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

- containers (templated data structures),
- iterators (for traversing containers and accessing their elements) and
- algorithms (which use iterators to manipulate containers).

12. In industrial-strength systems, you’ll use standard library classes for this, but this example enables us to go “under the hood” to demonstrate many key Modern C++ concepts.

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

This chapter presents the second of our two systems programming case study exercises—**Building Your Own Compiler**. In the context of several exercises, you'll build a simple compiler that translates programs written in a small high-level programming language into our Simpletron Machine Language (SML). You'll write programs in this small new high-level language, compile them on the compiler you build, then run them on your Simpletron virtual machine you built in Chapter 7's systems programming case study exercise—**Building Your Own Computer**. And with Chapter 8's file-processing techniques, your compiler can write the generated machine-language code into a file from which your Simpletron computer can then read your SML program, load it into the Simpletron's memory and execute it! This is a nice end-to-end systems-programming exercise sequence for novice computing students.

Chapter 14, Standard Library Algorithms and C++20 Ranges & Views, presents many of the standard library's 115 algorithms, focusing on the C++20 range-based algorithms, which are easier to use than their pre-C++20 versions. As you'll see, range-based algorithms specify their requirements using C++20 concepts—a C++20 “big four” feature that makes generic programming with templates more convenient and powerful. We briefly introduce C++20 concepts as needed for you to understand the requirements for working with these algorithms—Chapter 15 discusses concepts in more depth. Algorithms we present include filling containers with values, generating values, comparing elements or entire containers, removing elements, replacing elements, mathematical operations, searching, sorting, swapping, copying, merging, set operations, and calculating minimums and maximums. We discuss each algorithm's minimum iterator requirements so you can determine which containers can be used with each algorithm. We also continue our discussion of C++'s functional-style programming features with C++20 ranges and views.

Part 5: Advanced Topics

Chapter 15, Templates, C++20 Concepts and Metaprogramming, presents our substantially enhanced treatment of compile-time (static) polymorphism, generic programming with templates, C++20 concepts and template metaprogramming. The importance of templates has increased with each new C++ release. A major Modern C++ theme is to do more at compile-time for better type checking and better runtime performance—anything resolved at compile-time avoids runtime overhead and makes systems faster. As you'll see, templates and especially template metaprogramming are the keys to powerful compile-time operations.

We demonstrate C++20's new template capabilities, including abbreviated function templates, templated lambdas and concepts. We introduce type traits for testing type attributes at compile-time. We show variadic function templates that receive a variable number of parameters and use fold expressions to conveniently apply an operation to all the arguments passed to a variadic template.

Concepts 

Perf 

Concepts 

A feature of this chapter is the **Crafting Valuable Classes case study**—you’ll reimplement Chapter 11’s **MyArray case study** as a class template with custom iterators that enable most C++ standard library algorithms to manipulate **MyArray** objects. We also define a custom algorithm that can process **MyArray** elements and standard library container class objects. We show that you can use concepts to overload functions based on the type requirements of their parameters. Finally, we introduce template metaprogramming for performing compile-time calculations, enabling you to improve a program’s execution-time performance, possibly reducing both execution time and memory consumption.

Chapter 16, C++20 Modules, presents another of C++20’s “big four” features. Modules provide a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details. Modules help you be more productive, especially when building, maintaining and evolving large software systems. Modules help such systems build faster and make them more scalable. 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).*”¹³ You’ll see that, even in small systems, modules offer immediate benefits in every program by eliminating the need for the C++ preprocessor. In several **Software Engineering case studies**, you’ll learn several ways to **separate interface from implementation using modules**.



Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, is the first of two extensive chapters on concurrency and multi-core programming. Chapter 17 is one of the most important chapters in the book, presenting C++’s features for building applications that create and manage multiple tasks. These can significantly improve program performance and responsiveness on today’s multi-core processors.



This chapter presents several **multithreading and multicore systems performance case studies**. In the **Profiling Sequential and Parallel Sorting Algorithms case study**, we show how to use prepackaged parallel algorithms to create multithreaded programs that will run faster (often much faster) on today’s multi-core computer architectures. For example, we sort 100 million values using a sequential sort, then a parallel sort. We use **timing operations from C++’s <chrono> library features to profile the performance improvement we get on today’s popular multi-core systems, as we employ more cores**. You’ll see that the parallel sort runs 6.76 times faster than the sequential sort on our computer with an 8-core Intel processor.

In the **Producer–Consumer: Synchronizing Access to Shared Mutable Data case studies**, we discuss the producer–consumer relationship and demonstrate various ways to implement it using low-level and high-level C++ concurrency primitives. We also present several **Coordinating Threads case studies using C++20’s new latch, barrier and semaphore capabilities**. We emphasize that concurrent programming is difficult to get right, so you should prefer the easier-to-use, higher-level concurrency features. Lower-level features like semaphores and atomics can be used to implement higher-level features like latches.

Chapter 18, C++20 Coroutines, is the second of our chapters on concurrency and multi-core programming. This chapter presents coroutines—the last of C++20’s “big four” features. **A coroutine is a function that can suspend its execution and be resumed later**,

13. Bjarne Stroustrup, “Modules and Macros.” February 11, 2018. Accessed March 8, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf>.

enabling you to do concurrent programming with a simple sequential-like coding style. The mechanisms supporting this are handled entirely by code that's written for you by the compiler. A function containing any of the keywords `co_await`, `co_yield` or `co_return` is a coroutine.



Coroutines require sophisticated infrastructure, which you can write yourself, but doing so is complex, tedious and error-prone. Instead, most experts agree that you should use high-level coroutine support libraries, which is the approach we show. The open-source community has created several experimental libraries for developing coroutines quickly and conveniently—we use two in our presentation. C++23 has standard library support for generator coroutines, and more coroutine support is expected in C++26.



We present three multithreading and multicore systems performance case studies:

- Creating a Generator Coroutine with `co_yield` and the `generator` Library
- Launching Tasks with `concurrentpp`
- Creating a Coroutine with `co_await` and `co_return`

To help readers understand how coroutines work, the first and third case studies include diagrams illustrating each application's flow of control.

Chapter 19, Stream I/O & C++20 Text Formatting, discusses C++ stream input/output capabilities and formatting features. We include stream formatting primarily for people who might encounter it in legacy code. Section 19.9 presents an in-depth case study on C++20's new text-formatting features. In introductory computer science courses, instructors may wish to present Section 19.9 after we introduce C++20 text formatting in Chapter 4.

Chapter 20, Other Topics and a Look Toward the Future of C++, continues our discussion of runtime polymorphism from Chapter 10 with case studies on runtime type information (RTTI), inheriting base-class constructors, the non-virtual interface idiom, duck typing with `std::variant` and `std::visit` (for runtime polymorphism with objects of classes that are not related by inheritance), and multiple inheritance. The chapter also presents miscellaneous C++ topics, including storage classes, storage duration, `mutable` class members, namespaces, operator keywords, pointers to class members, the `[[nodiscard]]` attribute, the `std::shared_ptr` and `std::weak_ptr` smart pointers, determining types at compile-time with `decltype`, and the `[[likely]]` and `[[unlikely]]` attributes. The chapter ends with a look forward to features coming in the C++23 and C++26 standards.

Chapter 21, Computer Science Thinking: Searching, Sorting and Big O, introduces some classic computer-science topics. We consider several algorithms and compare their processor demands and memory consumption. We present a friendly introduction to computer science's Big *O* notation, which indicates how hard an algorithm may have to work to solve a problem based on the number of items it must process.



The chapter includes two case studies that visualize the high-speed binary search and merge sort algorithms to illustrate how these algorithms work. In introductory computer science courses, instructors can present this chapter after Chapter 6.

Modern C++ Data Structures Courses

Our recursion (Chapter 5), arrays (Chapter 6), searching (Chapters 6 and 21), sorting (Chapters 6 and 21), Big *O* (Chapter 21), containers (Chapter 13), iterators (Chapter 13)

and algorithms (Chapter 14) coverage provides nice content for a course emphasizing Modern C++ data structures.

7 Compilers, Docker and Static Code Analysis Tools

Industrial-Strength Compilers

We tested all the code for correctness on the Windows, macOS and Linux operating systems using the latest versions of

- Visual C++® in Microsoft® Visual Studio® Community edition on Windows®,
- GNU® C++ (`g++`) and
- Clang C++ (`clang++`).

See the Before You Begin section that follows this Preface for software installation instructions.

Most C++20 features are now fully implemented in these compilers. We point out exceptions as appropriate. As coverage improves, we'll post code updates to the book's GitHub repository:

<https://github.com/pdeitel/CPlusPlusHowToProgram11e>

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

<https://www.deitel.com/books/cpphtp11>

At the time of this writing, Apple's Xcode integrated development environment (IDE) did not support various key C++20 features we use. Once these features become available in Xcode, we'll post Xcode instructions on the preceding website.

Docker

Docker is a tool for packaging software into containers that bundle everything required to execute that software conveniently and portably across platforms. Docker provides a simple way to help you get started with new technologies quickly, conveniently and economically on your desktop or notebook computers. We show how to install and execute Docker containers preconfigured with

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

Each can run in Docker on Windows, macOS and Linux, enabling users to try the latest versions of these compilers. Chapter 1 includes test-drives showing how to compile programs and run them in the context of cross-platform Docker containers.

Static Code Analysis Tools

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

- clang-tidy—<https://clang.llvm.org/extrac clang-tidy/>
- cppcheck—<https://cppcheck.sourceforge.io/>



- Microsoft’s C++ Core Guidelines static code analysis tools, which are built into Visual Studio’s static code analyzer

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

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

We also used the compiler flag `-Wall` in the GNU `g++` and LLVM `clang++` compilers to enable all compiler warnings. Most of our programs compile without warning messages. See the Before You Begin section that follows this Preface for information on configuring the C++ Core Guidelines checker in Microsoft Visual C++.

8 Thinking Like a Developer—GitHub, StackOverflow and More

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

You’ll work with such popular websites as GitHub and StackOverflow, and you’ll do lots of Internet research.

- StackOverflow is one of the most popular programming question-and-answer sites. Many problems you might encounter have already been discussed here. It’s a great place to ask code-oriented questions. Many of our Google searches for various, often complex, issues throughout our writing effort returned StackOverflow posts as their first results.
- GitHub is an excellent venue for finding free, open-source code to explore and incorporate into your projects—and for you to contribute your code to the open-source community if you like. One hundred million developers use GitHub.¹⁵ The site hosts over 330 million repositories for code in many programming languages¹⁶—developers made 413 million contributions to repositories in 2022.¹⁷

14. William Gates, quoted in *Programmers at Work: Interviews With 19 Programmers Who Shaped the Computer Industry* by Susan Lammers. Microsoft Press, 1986, p. 83.

15. “Let’s build from here: The complete developer platform to build, scale, and deliver secure software.” Accessed March 8, 2023. <https://github.com/about>.

16. “Let’s build from here: The complete developer platform to build, scale, and deliver secure software.” Accessed March 8, 2023. <https://github.com/about>.

17. “OCTOVERSE 2022: The state of open source software.” Accessed March 8, 2023. <https://octoverse.github.com>.

GitHub is a crucial element of the professional software developer’s arsenal, with version-control tools that help developer teams manage public open-source projects and private projects. There is a massive C++ open-source community on GitHub where developers contribute to almost 58,000¹⁸ C++ code repositories. We encourage you to study and execute lots of developers’ open-source C++ code. This is a great way to learn and is a natural extension of our live-code teaching approach.¹⁹

9 Computing and Data Science Curricula



This book is designed for courses that adhere to one or more of the following ACM/IEEE CS-and-related curricula, which call for covering security, data science, ethics, privacy and performance concepts and using real-world data:

- Computer Science Curricula 2013,²⁰
- CC2020: A Vision on Computing Curricula,²¹
- Computing Curricula 2020 recommendations,²²
- Information Technology Curricula 2017,²³
- Cybersecurity Curricula 2017,²⁴
- the 2016 data science initiative “Curriculum Guidelines for Undergraduate Programs in Data Science”²⁵ from the faculty group sponsored by the NSF and the Institute for Advanced Study, and
- ACM Data Science Task Force’s Computing Competencies for Undergraduate Data Science Curricula Final Report.²⁶

18. “C++.” Accessed March 8, 2023. <https://github.com/topics/cpp>.

19. You’ll need to become familiar with the variety of open-source licenses for software on GitHub.

20. ACM/IEEE (Assoc. Comput. Mach./Inst. Electr. Electron. Eng.). 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* (New York: ACM). Accessed March 8, 2023. <http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-final-report.pdf>.

21. A. Clear, A. Parrish, G. van der Veer and M. Zhang “CC2020: A Vision on Computing Curricula.” Accessed March 8, 2023. <https://dl.acm.org/citation.cfm?id=3017690>.

22. CC2020 Task Force, *Computing Curricula 2020*. Accessed March 8, 2023. <https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf>.

23. *Information Technology Curricula 2017*. Accessed March 8, 2023. <http://www.acm.org/binaries/content/assets/education/it2017.pdf>.

24. *Cybersecurity Curricula 2017*. Accessed March 8, 2023. https://cybered.hosting.acm.org/wp-content/uploads/2018/02/newcover_csec2017.pdf.

25. “Curriculum Guidelines for Undergraduate Programs in Data Science” Accessed March 8, 2023. <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930>.

26. ACM Data Science Task Force, *Computing Competencies for Undergraduate Data Science Curricula*. Accessed March 8, 2023. https://dstf.acm.org/DSTF_Final_Report.pdf.

Computing Curricula

- According to “CC2020: A Vision on Computing Curricula,”²⁷ the curriculum “needs to be reviewed and updated to include the new and emerging areas of computing such as cybersecurity and data science.”²⁸



10 Data Science Overlaps with Computer Science²⁹

The undergraduate data science curriculum proposal³⁰ includes algorithm development, programming, computational thinking, data structures, database, mathematics, statistical thinking, machine learning, data science and more—a significant overlap with computer science, especially given that the data science courses include some key AI topics. We work some basic data science topics into various examples, exercises, projects and case studies.

Key Points from the Data Science Curriculum Proposal

This section calls out some key points from the data science undergraduate curriculum proposal and its detailed course descriptions appendix.³¹ We cover each of the following:

- Learn programming fundamentals commonly presented in computer science courses, including working with data structures.
- Be able to solve problems by creating algorithms.
- Work with procedural, functional and object-oriented programming.
- Explore concepts via simulations.
- Use development environments (we tested all our code on Microsoft Visual C++, GNU g++ and LLVM clang++).
- Work with real-world data in practical case studies and projects.
- Create data visualizations.
- Work with existing software.
- Work with high-performance tools, such as C++’s multithreading libraries.
- Focus on data’s ethics, security and privacy issues.

11 Appendices on Deitel.com

On the textbook’s webpage at <https://deitel.com/cpphtp11>, we provide several appendices to support the book:

-
27. A. Clear, A. Parrish, G. van der Veer and M. Zhang, “CC2020: A Vision on Computing Curricula,” <https://dl.acm.org/citation.cfm?id=3017690>.
 28. <http://delivery.acm.org/10.1145/3020000/3017690/p647-clear.pdf>.
 29. This section is intended primarily for data science instructors but includes important information for computer science instructors as well.
 30. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930>.
 31. “Appendix—Detailed Courses for a Proposed Data Science Major,” http://www.annualreviews.org/doi/suppl/10.1146/annurev-statistics-060116-053930/suppl_file/st04_de_veaux_supmat.pdf.

- Appendix A, Character Set, contains the letters, digits and symbols of the ASCII character set.
- Appendix B, Number Systems, overviews the binary, octal, decimal and hexadecimal number systems.
- Appendix C, Preprocessor, discusses additional features of the C++ preprocessor. Template metaprogramming (Chapter 15) and C++20 Modules (Chapter 16) eliminate the need for many of this appendix's features.
- Appendix D, Bit Manipulation, discusses bitwise operators for manipulating the individual bits of integral operands and bit fields for compactly representing integer data.

Other Web-Based Materials on deitel.com

The book's webpage also contains:

- Links to our GitHub repository containing the downloadable C++ source code
- Blog posts—<https://deitel.com/blog>
- Book updates

For more information about downloading the code examples and setting up your C++ development environment, see the Before You Begin section that follows this Preface.

12 C++ Core Guidelines

The C++ Core Guidelines



<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

are recommendations “to help people use modern C++ effectively.”³² They’re edited by Bjarne Stroustrup (C++’s creator) and Herb Sutter (Convener of the ISO C++ Standards Committee). According to the overview:

“The guidelines are focused on relatively high-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast—you can afford to do things right.”³³



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

32. C++ Core Guidelines, “Abstract.” Accessed March 8, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract>.

33. C++ Core Guidelines, “Abstract.”

Guidelines Support Library

The C++ Core Guidelines often refer to capabilities of the Guidelines Support Library (GSL), which provides reusable C++ components that support various recommendations.³⁴ Microsoft provides an open-source GSL implementation on GitHub at

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

For your convenience, we include with the book's code examples the version of this library that we used in a few examples. Some GSL features have been incorporated into the C++ standard library.

13 Pedagogic Features and Conventions

 *C++ How to Program: An Objects-Natural Approach, 11/e* contains hundreds of live-code examples. We stress program clarity and concentrate on building well-engineered software.

Using Fonts for Emphasis

Our C++ code uses a fixed-width font (e.g., `x = 5`). We place on-screen components in the **bold Helvetica** font (e.g., the **File** menu).

Syntax Coloring

For readability, we syntax color all the code. Our e-book syntax-coloring conventions are:

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

Objectives and Outline

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

Tables and Illustrations

Abundant tables and line drawings are included.

Programming Tips and Key Features

We call out programming tips and key features with icons in the margins (see Section 5).

Index

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

C++ Programming Fundamentals

In our rich coverage of C++ fundamentals:

- We emphasize problem-solving and algorithm development.
- To help students prepare to work in industry, we use the terminology from the latest C++ standard document in preference to general programming terms.

34. C++ Core Guidelines, “GSL: Guidelines Support Library.” Accessed March 8, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl>.

- We avoid heavy math, leaving it to upper-level courses. Optional mathematical exercises and projects are included for science and engineering courses.

Innovation: “Intro-to” Pedagogy with 452 Integrated Checkpoint Exercises

This book uses our new “Intro to” pedagogy with integrated Checkpoint exercises and answers. We introduced this pedagogy in our textbook, *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud* (<https://deitel.com/pycds>). Chapter sections are intentionally small. We use a “read-a-little, code-a-little, test-a-little” approach. In the core computer science chapters (1–12 and 21), you read about a new concept, study and execute the corresponding code examples, then test your understanding via the integrated fill-in-the-blank, true/false, discussion and code-based Checkpoint exercises immediately followed by their answers. This will help you keep a brisk learning pace.



KIS (Keep It Simple), KIS (Keep it Small), KIT (Keep it Topical)

- Keep it simple—We strive for simplicity and clarity.
- Keep it small—Many of the book’s examples are small. We use more substantial code examples, exercises and projects when appropriate, particularly in the case studies that are a core feature of this textbook.
- Keep it topical—to “take the pulse” of Modern C++, which changes the way developers write C++ programs, we read, browsed or watched approximately 6,000 current articles, research papers, white papers, books, documentation pieces, blog posts, forum posts, webinars and videos.
- We show C++ as it’s intended to be used with a rich collection of applications programming and systems programming case studies, focusing on computer science, artificial intelligence, data science and other fields.



Over 700 Contemporary Examples, Exercises and Projects (EEPs)

Consistent with our live-code and object-natural approaches, you’ll learn hands-on from a broad selection of 255 real-world examples and case studies, and 476 exercises and projects drawn from computer science, data science and other fields:



- Our code examples, exercises and projects familiarize students with current topics of interest to developers. We begin in Chapter 1 by briefly touring topics of current interest including open-source software, virtualization, simulation, web services, multithreading, multicore hardware architecture, systems programming, artificial intelligence, natural language processing, data science, robust secure programming, cryptography, Docker, GitHub, StackOverflow, forums, the metaverse, blockchain, NFTs (nonfungible tokens), cryptocurrencies (like Bitcoin and Ethereum), generative AI (ChatGPT, Dall-E), general artificial intelligence and more.
- We added a variety of systems programming and application programming case studies. Some are book sections that walk through the complete source code, some are exercises with detailed specifications from which you should be able to develop the code solution on your own, and some are exercises requiring additional research. We enumerate the 50 case studies and case-study exercises in this preface.



- You'll attack exciting and entertaining challenges in our larger case studies, such as building a casino game, building your own computer (using simulation to build a virtual machine), using AI/data-science technologies such as basic natural language processing, building your own compiler, writing multithreaded code to take advantage of today's multicore computer architectures to get the best performance from your computer and many more.
- Research and project exercises ask you to go deeper into what you've learned and explore other technologies. We encourage you to use computers and the Internet to solve significant problems. Projects are often more elaborate than the exercises—some might require days or weeks of implementation effort. Many are appropriate for class projects, term projects, directed-study courses, capstone-course projects and thesis research. We do not provide solutions for the projects.
- We've enhanced existing case studies and added new ones focusing on AI and data science, including simulations with random-number generation, Anscombe's Quartet, natural language processing (NLP) and artificial intelligence via heuristic programming.
- Instructors can tailor their courses to their audience's unique requirements and vary labs and exam questions each semester.



Extensive Videos

In the Pearson interactive eText and Revel versions of this book, we provide extensive videos in which Paul Deitel discusses the material in the Before You Begin section and Chapters 1–10.

Glossary Items

In the Pearson interactive eText and Revel versions of the book, we added over 300 glossary items for the core computer science chapters (1–12 and 21). These are also used in student learning tools to create flashcards and matching exercises.



Performance

Software developers prefer C++ (and C) for performance-intensive operating systems, real-time systems, embedded systems, game systems and communications systems, so we focus on performance issues.



Security Emphasis and Cryptography Case Studies

Consistent with our richer treatment of security, we've added case studies on secret-key and public-key cryptography. The latter is a project exercise that includes a detailed walk-through of the enormously popular RSA algorithm's steps, providing hints to help you build a working, simple, small-scale classroom implementation.

Working with Open-Source Software

Open source is software with source code that anyone can inspect, modify, and enhance.³⁵ We encourage you to try lots of demos and view free, open-source code examples (available on sites such as GitHub) for inspiration.

35. "What is open source?" Accessed March 8, 2023. <https://opensource.com/resources/what-open-source>.



Data Experiences

In Chapter 9, you'll work with real-world text data. You'll read and analyze the Titanic Disaster dataset—popular for introducing data analytics. Datasets are often stored in CSV (comma-separated values) files, which we introduce in Chapter 9. You'll also download and analyze Shakespeare's play *Romeo and Juliet* and Christopher Marlowe's play *Edward the Second* from Project Gutenberg—a source of free downloadable texts for analysis. The site contains over 60,000 e-books in various formats, including plain-text files—these are out of copyright in the United States.

Privacy

The ACM/IEEE's curricula recommendations³⁶ for Computer Science, Information Technology and Cybersecurity mention privacy over 200 times. Every programming student and professional needs to be acutely aware of privacy issues and concerns. Students research privacy in four exercises in Chapters 1 and 3. We also discuss cryptography, which is critical in maintaining privacy, in Chapters 5 and 10.

In Chapter 1's exercises, you'll start thinking about these issues by researching ever-stricter privacy laws such as HIPAA (Health Insurance Portability and Accountability Act), the California Consumer Privacy Act (CCPA) in the United States and GDPR (General Data Protection Regulation) for the European Union.

Ethics

The ACM's curricula recommendations³⁷ for Computer Science, Information Technology and Cybersecurity mention ethics more than 100 times. In several Chapter 1 exercises, you'll focus on ethics issues via Internet research. You'll investigate privacy and ethical issues surrounding intelligent assistants, such as Amazon Alexa, Apple Siri, Google Assistant and Microsoft Cortana. And we'll look at the excitement and controversy surrounding OpenAI's ChatGPT³⁸ and Dall-E 2.³⁹

14 Instructor Supplements

The following supplements are available only to qualified instructors through Pearson Education's Instructor Resource Center (<https://pearsonhighered.com/irc>):

- The Instructor Solutions Manual contains solutions to most of the end-of-chapter exercises. Solutions are not provided for “project” exercises.
- A Test Item File containing multiple-choice questions and answers.
- Lecture slides containing diagrams, tables and bulleted items summarizing key points in the text.

36. “Curricula Recommendations.” Accessed March 8, 2023. <https://www.acm.org/education/curricula-recommendations>.

37. “Curricula Recommendations.” Accessed March 8, 2023. <https://www.acm.org/education/curricula-recommendations>.

38. “Introducing ChatGPT.” Accessed March 8, 2023. <https://openai.com/blog/chatgpt>.

39. “Dall-E 2.” Accessed March 8, 2023. <https://openai.com/product/dall-e-2>.

The lecture slides do not include the source code—the source-code files⁴⁰ for the hundreds of live-code examples are available to instructors and students in the book’s GitHub repository at

<https://github.com/pdeitel/CPlusPlusHowToProgram11e>

If you’re not a GitHub user, click the green **Code** button and select **Download ZIP** to download a ZIP file containing the code. See the Before You Begin section for more information.

Please do not write to us requesting access to the Pearson Instructor’s Resource Center. Access is restricted to college instructors who have adopted the book for their courses. Instructors may obtain access only through their Pearson representatives. If you’re not a registered faculty member, contact your Pearson representative or visit

<https://pearson.com/relocator>

15 Some Key C++ Documentation and Resources

The book includes almost 800 citations to videos, blog posts, articles, whitepapers and online documentation we studied while writing the manuscript. You may want to access some of these resources to investigate more advanced features and idioms. The website cppreference.com has become the defacto C++ documentation site. We reference it frequently so you can get more details about the standard C++ classes and functions we use throughout the book. We also frequently cite the final draft of the C++20 standard document, which is available free on GitHub at

<https://timsong-cpp.github.io/cppwp/n4861/>

The C++ standard committee’s evolving working draft (currently C++23) is available at:

<https://eel.is/c++draft/>

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

Documentation

- C++ Reference at <https://cppreference.com/>
- Microsoft’s C++ language documentation: <https://docs.microsoft.com/en-us/cpp/>
- The GNU C++ Standard Library Reference Manual: <https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html>

16 Getting Your Questions Answered

Popular C++ and general programming online forums include

- <https://stackoverflow.com>
- <https://www.reddit.com/r/cpp/>
- <https://groups.google.com/g/comp.lang.c++>

40. We recommend that instructors present source code in their favorite C++ integrated development environments (IDEs) or code-oriented text editors. These typically provide customizable syntax highlighting and adjustable font sizes for presentation purposes.

For a list of other valuable sites, see

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

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

Communicating with the Authors

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

deitel@deitel.com

We'll respond promptly.

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

You can keep up-to-date with Deitel on the following social media platforms:

- LinkedIn®—<https://www.linkedin.com/company/deitel-&-associates/>
- YouTube®—<https://youtube.com/DeitelTV>
- Twitter®—<https://twitter.com/deitel>
- Facebook®—<https://facebook.com/DeitelFan>
- Instagram®—<https://instagram.com/DeitelFan>

18 Live Instructor-Led Training with Paul Deitel

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

19 College Textbook Digital Formats

Our college textbook, *C++ How to Program: An Objects-Natural Approach, 11/e*, is available in three digital formats:

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

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

- A chapter introducing hardware, software and Internet concepts.
- An introduction to programming for novices.

- End-of-section programming and non-programming Checkpoint self-review exercises with answers.
- End-of-chapter exercises.

Deitel Pearson eTexts and Revels include:

- Videos in which Paul Deitel discusses the material in the book's core CS1 chapters (1–10).
- Interactive programming and non-programming Checkpoint self-review exercises with answers.
- Glossaries, flashcards, matching exercises and other learning tools.

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

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

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

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

20 Acknowledgments

We'd like to thank Barbara Deitel for long hours devoted to Internet research on this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the guidance, wisdom, energy and editorial savvy of Tracy Johnson (Pearson Education, Global Content Manager, Computer Science)—on all our academic publications. She challenges us at every step of the process to “get it right” and make the best books. Bob Engelhardt managed the book's production. Erin Sullivan recruited and managed the academic reviewers; Charvi Arora recruited and managed the professional reviewers. We selected the cover art, and Chuti Prasertsith designed the cover, adding his special touch of graphics magic.

Reviewers

We were fortunate on this project to have 14 distinguished academics and professionals review the manuscript. Most of the professional reviewers are either on the ISO C++ Standards Committee, have served on it or have a working relationship with it. Many have contributed features to the language. They helped us make a better book—any remaining flaws are our own.

Academic Review Team—Prof. Jeffrey Davis, School of Electrical and Computer Engineering, Georgia Institute of Technology; M. Michael Hadavi, The MathWorks, Inc., MET Computer Science at Boston University; Dr. Ningfang Mi, Associate Professor

of Electrical and Computer Engineering, Northeastern University; Prof. Patrice Roy, Université de Sherbrooke, ISO C++ Standards Committee Member.

Professional Review Team—Andreas Fertig, Independent C++ Trainer and Consultant, Creator of `cppinsights.io`, Author of *Programming with C++20*; Marc Gregoire, Software Architect, Nikon Metrology, Microsoft Visual C++ MVP and author of *Professional C++, 5/e*; Dr. Daisy Hollman, ISO C++ Standards Committee Member; Danny Kalev, Ph.D. and Certified System Analyst and Software Engineer, Former ISO C++ Standards Committee Member; Dietmar Kühl, Senior Software Developer, Bloomberg L.P., ISO C++ Standard Committee Member; Inbal Levi, SolarEdge Technologies, ISO C++ Foundation director, ISO C++ SG9 (Ranges) chair, ISO C++ Standards Committee member; Arthur O'Dwyer, C++ trainer, Chair of CppCon's Back to Basics track, author of several accepted C++17/20/23 proposals and the book *Mastering the C++17 STL*; Saar Raz, Senior Software Engineer, Swimm.io and Implementor of C++20 Concepts in Clang; José Antonio González Seco, Parliament of Andalusia; Anthony Williams, Member of the British Standards Institution C++ Standards Panel, Director of Just Software Solutions Ltd., Author of *C++ Concurrency in Action, 2/e* (Anthony is the author or co-author of many C++ Standard Committee papers that led to C++'s standardized concurrency features).

Google Search

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

Grammarly

We use the paid version of Grammarly on all our manuscripts. They describe their tools as helping you “compose bold, clear, mistake-free writing” with their “AI-powered writing assistant.”⁴¹ Grammarly also provides free tools that you can integrate into several popular web browsers, Microsoft® Office 365™ and Google Docs™.

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

`deitel@deitel.com`

We'll respond promptly.

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

*Paul Deitel
Harvey Deitel*

41. “Grammarly.” Accessed March 8, 2023. <https://www.grammarly.com>.

21 About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is an MIT graduate with 43 years in computing. He is one of the world's most experienced programming-languages trainers, having taught professional courses to software developers since 1992. He has delivered hundreds of programming courses to academic, industry, government and military clients of Deitel & Associates, Inc. internationally, including UCLA, SLB (formerly Schlumberger), Cisco, IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Puma, iRobot and many more.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 62 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science departments. He has extensive college and professional teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with more than 100 translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to academic, corporate, government and military clients.

22 About Deitel® & Associates, Inc.

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

Through its 48-year publishing partnership with Pearson, Deitel & Associates, Inc. publishes leading-edge computer programming college textbooks and professional books in print and digital formats, LiveLessons video courses, O'Reilly Online Learning live training courses and Revel™ and eText interactive multimedia college courses.

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

deitel@deitel.com

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

<https://deitel.com/training>

Individuals wishing to purchase Deitel books can do so at

<https://amazon.com>

<https://www.barnesandnoble.com/>

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

`corpsales@pearsoned.com`

Deitel e-books are available in various formats from

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

`https://www.vitalsource.com/`

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

`https://www.redshelf.com/`

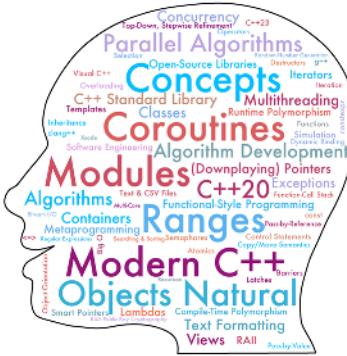
`https://www.informit.com/`

`https://www.chegg.com/`

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

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

This page intentionally left blank



Before You Begin

Before using this book, please read this section to understand our conventions and set up your computer to compile and run our example programs. If there are changes to the instructions presented here, we'll post updates on the book's webpage:

<https://deitel.com/cpphtp11>

Font and Naming Conventions

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

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

Obtaining the Code Examples

Download the *C++ How to Program: An Objects-Natural Approach, 11/e* code examples from our GitHub repository at

<https://github.com/pdeitel/CPlusPlusHowToProgram11e>

If you're familiar with Git and GitHub, clone the repository to your system. If you're not a GitHub user, click the green **Code** button and select **Download ZIP** to download a ZIP file containing the code. To extract the ZIP file's contents:

- Windows: Right-click the ZIP file, select **Extract All...**, select your user account's **Documents** folder, then click **Extract**.
- macOS: Move the ZIP file to your user account's **Documents** folder, then double-click the ZIP file.
- Linux (varies by distribution): When you download the ZIP file on Ubuntu Linux, you can choose to open the file with the **Archive Manager** or save it. Choose **Archive Manager**, then click **Extract** in the window that appears. Select your user account's **Documents** folder, then click **Extract** again.

Throughout the book, our instructions assume the code examples reside in your user account's **Documents** folder in a subfolder named **examples**.

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

<https://guides.github.com/activities/hello-world/>

Compilers We Use

Ensure that you have a recent C++ compiler installed. We tested the book's code examples using the following free compilers:

- For Microsoft Windows, we used Microsoft Visual Studio Community edition, which includes the Visual C++ compiler and other Microsoft development tools.
- For Linux, we used the GNU C++ compiler (`g++`)¹—part of the GNU Compiler Collection (GCC). Typically, a version of GNU C++ is pre-installed on most Linux systems. You might need to update the compiler to a more recent version. GNU C++ also can be installed on macOS and Windows systems.
- For macOS, we used both the GNU C++ compiler (`g++`) and the Apple Xcode² C++ compiler, which uses a version of the LLVM Clang C++ compiler (`clang++`).
- You can run the latest versions of GNU C++ (`g++`) and LLVM Clang C++ (`clang++`)³ conveniently on Windows, macOS and Linux via Docker containers. See the “Docker and Docker Containers” section in this Before You Begin section.

At the time of this writing, Apple Xcode does not support several key C++20 features we use throughout this book, so we recommend using the most recent version of `g++`. When Xcode’s C++20 support changes, we’ll post updates at

<https://deitel.com/cpphtp11>

This Before You Begin describes installing the compilers and Docker. Section 1.11’s test-drives demonstrate how to compile and run C++ programs using these compilers.

Installing Visual Studio Community Edition on Windows

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

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

Next, go to

<https://visualstudio.microsoft.com/downloads/>

Then perform the following installation steps:

1. Click **Free Download** under **Community**.
2. Depending on your web browser, you may see a pop-up at the bottom of your screen where you can click **Run** to start the installation process. If not, double-click the installer file in your **Downloads** folder when the download completes.
3. In the **User Account Control** dialog, click **Yes** to allow the installer to make changes to your system.
4. In the **Visual Studio Installer** dialog, click **Continue** to allow the installer to download the components it needs for you to configure your installation.

1. GNU C++ (`g++`) 13.1 at the time of this writing.

2. Xcode 14.3.1 at the time of this writing.

3. Clang C++ (`clang++`) 16 at the time of this writing.

5. For this book's examples, select the option **Desktop Development with C++**, which includes the Visual C++ compiler and the C++ standard libraries.
6. Click **Install**. The installation process can take a significant amount of time.

Installing Xcode on macOS

On macOS, perform the following steps to install Xcode:

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

Installing GNU C++ (g++) 13 on macOS

On macOS, perform the following steps to install GNU C++:

1. In the **Finder**'s **Go** menu, select **Utilities**, then double-click **Terminal** to open a **Terminal** (command line) window.
2. Check if the **brew** command is installed by typing **brew** and pressing *Enter* (or *return*). If macOS does not recognize the command, go to <https://brew.sh> and copy the installation command below **Install Homebrew**. Paste this command into the **Terminal** window, then press *Enter* (or *return*).
3. Type the following command, then press *Enter* (or *return*) to install the GNU Compiler Collection (GCC), which includes **g++**:

```
brew install gcc@13
```

Installing the GNU C++ (g++) 13 on Linux

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

<https://gcc.gnu.org/install/binaries.html>

Docker and Docker Containers

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

Installing Docker

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

<https://www.docker.com/get-started>

Then follow the on-screen instructions. Also, sign up for a **Docker Hub** account on this site, which gives you access to the many containers at <https://hub.docker.com>. Linux users should install **Docker Engine** from

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

Getting the GNU Compiler Collection Docker Container

The GNU team maintains official Docker containers at

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

Once Docker is installed and running, open a Command Prompt⁴ (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command

```
docker pull gcc:latest
```

Docker downloads a container configured with the GNU Compiler Collection (GCC)'s most current version—13.1 at the time of this writing. In one of Section 1.11's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

Getting an LLVM Clang C++ Docker Container

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

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

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

```
docker pull teeks99/clang-ubuntu:16
```

Docker downloads a container configured with LLVM Clang's most current version—16 at the time of this writing. In one of Section 1.11's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

Getting Your C++ Questions Answered

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

deitel@deitel.com

and

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

We'll respond promptly.

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

<https://stackoverflow.com>

on which you can

- search for answers to common programming questions,
- search for error messages to see what causes them,

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

- ask programming questions to get answers from programmers worldwide and
- gain valuable insights about programming in general.

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

<https://cpplang-inviter.cppalliance.org>

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

<https://www.includecpp.org/discord/>

Online C++ Documentation

For C++ standard library documentation, visit

<https://cppreference.com>

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

<https://isocpp.org/faq>

Static Code Analysis Tools

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

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

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

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

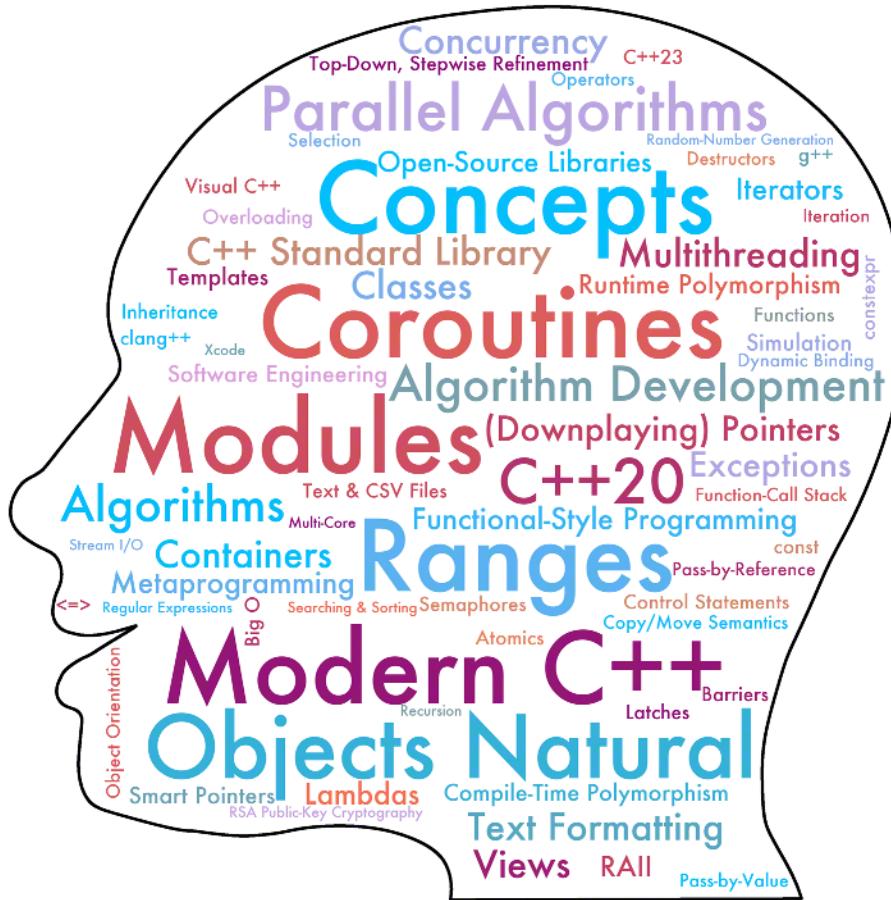
You can install cppcheck for various operating-system platforms by following the instructions at <https://cppcheck.sourceforge.io/>.

For Visual C++, once you learn how to create a project in Section 1.11's test-drives, you can configure Microsoft's C++ Core Guidelines static code analysis tools as follows:

1. Right-click your project name in the **Solution Explorer** and select **Properties**.
2. In the dialog that appears, select **Code Analysis > General** in the left column, then set **Enable Code Analysis on Build** to **Yes** in the right column.
3. Next, select **Code Analysis > Microsoft** in the left column. Then, in the right column, you can select a subset of the analysis rules from the drop-down list. We used the option **<Choose multiple rule sets...>** to select all the rules that begin with **C++ Core Check**. Click **Save As...**, give your custom rule set a name, click **Save**, then click **Apply**.

This page intentionally left blank

Intro to Computers and C++



Objectives

In this chapter, you'll:

- Learn computer hardware, software and Internet basics.
 - Understand a data hierarchy from bits to databases.
 - Understand the types of programming languages.
 - Understand the strengths of C++ and other languages.
 - Be introduced to the C++ standard library of reusable components.
 - Compile and run a C++ application using our three preferred compilers and Docker containers.
 - Be introduced to object-technology concepts used in the objects-natural case studies (Chapters 2–9) and discussed in detail in the object-oriented programming Chapters 9–11.
 - Understand how concurrent programming helps maximize performance on multi-core processors.
 - Be introduced to big data and data science.
 - Learn about exciting recent developments in computing, including the Metaverse, artificial intelligence and related technologies.

Outline

2 Chapter 1 Intro to Computers and C++

1.1 Introduction	
1.2 Hardware	
1.2.1 Computer Organization	
1.2.2 Moore's Law, Multi-Core Processors and Concurrent Programming	
1.3 Data Hierarchies	
1.4 Machine Languages, Assembly Languages and High-Level Languages	
1.5 Operating Systems	
1.6 C and C++	
1.7 C++ Standard Library and Open-Source Libraries	
1.8 Other Popular Programming Languages	
1.9 Introduction to Object Orientation	
1.10 Simplified View of a C++ Development Environment	
1.11 Test-Driving a C++20 Application Various Ways	
1.11.1 Compiling and Running on Windows with Visual Studio Community Edition	
1.11.2 Compiling and Running with GNU C++ on Linux	
1.11.3 Compiling and Running with <code>g++</code> in the GCC Docker Container	
1.11.4 Compiling and Running with <code>clang++</code> in a Docker Container	
1.11.5 Compiling and Running with Xcode on macOS	
1.12 Internet, World Wide Web, the Cloud and IoT	
1.12.1 The Internet: A Network of Networks	
1.12.2 The World Wide Web: Making the Internet User-Friendly	
1.12.3 The Cloud	
1.12.4 The Internet of Things (IoT)	
1.12.5 Edge Computing	
1.12.6 Mashups	
1.13 Metaverse	
1.13.1 Virtual Reality (VR)	
1.13.2 Augmented Reality (AR)	
1.13.3 Mixed Reality	
1.13.4 Blockchain	
1.13.5 Bitcoin and Cryptocurrency	
1.13.6 Ethereum	
1.13.7 Non-Fungible Tokens (NFTs)	
1.13.8 Web3	
1.14 Software Development Technologies	
1.15 How Big Is Big Data?	
1.15.1 Big-Data Analytics	
1.15.2 Data Science and Big Data Are Making a Difference: Use Cases	
1.16 AI—at the Intersection of Computer Science and Data Science	
1.16.1 Artificial Intelligence (AI)	
1.16.2 Artificial General Intelligence (AGI)	
1.16.3 Artificial Intelligence Milestones	
1.16.4 Machine Learning	
1.16.5 Deep Learning	
1.16.6 Reinforcement Learning	
1.16.7 Generative AI—ChatGPT and Dall-E 2	
1.17 Wrap-Up	
Exercises	

1.1 Introduction

Welcome to C++—one of the world’s most senior computer programming languages and, according to various programming-language rankings, one of the world’s most popular.^{1,2,3} You’re probably familiar with many of the powerful tasks computers perform. This textbook provides an intensive, hands-on experience in which you’ll write C++ instructions that command computers to perform many of those and other tasks. **Software**—the C++ instructions you write, which also are called **code**—controls **hardware** (that is, computers and related devices).

-
1. “TIOBE Index.” Accessed March 18, 2023. <https://www.tiobe.com/tiobe-index/>.
 2. “Top Programming Languages 2022.” Accessed March 18, 2023. <https://spectrum.ieee.org/top-programming-languages-2022>.
 3. “PYPL PopularitY of Programming Language.” Accessed March 18, 2023. <https://pypl.github.io/PYPL.html>.

C++ is widely used in industry.⁴ Portions of today’s most popular desktop operating systems—Windows⁵ and macOS⁶—are written in C++. Many applications and systems are partially written in C++, including popular web browsers (e.g., Google Chrome⁷ and Mozilla Firefox⁸), database management systems (e.g., Microsoft SQL Server⁹, Oracle¹⁰, MySQL¹¹), Adobe’s creative applications, Amazon.com, Bloomberg, Facebook, various Microsoft apps (including Office and Visual Studio) and much more.¹²

The chapter introduces terminology and concepts that lay the groundwork for the C++ programming you’ll learn, beginning in Chapter 2. We introduce hardware and software concepts. We overview a sample data hierarchy—from bits (ones and zeros) to databases, which store the massive amounts of data that organizations need to implement contemporary applications such as Google Search, Netflix, Twitter, Waze, Uber, Airbnb and a myriad of others.

We discuss the types of programming languages. We introduce the C++ standard library and various C++ “open-source” libraries that help you avoid “reinventing the wheel”—you’ll write only modest numbers of C++ instructions to make these libraries perform powerful tasks.

We overview additional technologies you’ll likely use as you build software in your career. We discuss the Internet, the web, the cloud and the Internet of Things (IoT). We introduce the Metaverse and its related technologies—blockchain, cryptocurrencies, NFTs, Web3, augmented reality, virtual reality and mixed reality. Finally, we introduce big data, data science and artificial intelligence (AI) technologies—machine learning, deep learning and generative AIs like the recently introduced to much excitement ChatGPT and Dall-E 2.

Compilers

You can compile, build and run C++ applications in many development environments. You’ll work through one or more of Section 1.11’s test-drives showing how to compile and execute C++ code using:

- Microsoft Visual Studio Community edition for Windows.
- GNU g++ in a macOS Terminal window or Linux shell.
- LLVM clang++ in a macOS Terminal window or a Linux shell.

-
4. “C++.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B>.
 5. “Windows 11.” Wikipedia. Wikimedia Foundation, Accessed March 18, 2023. https://en.wikipedia.org/wiki/Windows_11.
 6. “macOS.” Wikipedia. Wikimedia Foundation, Accessed March 18, 2023. <https://en.wikipedia.org/wiki/MacOS>.
 7. “Google Chrome.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Google_Chrome.
 8. “Firefox.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Firefox>.
 9. “Microsoft SQL Server.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Microsoft_SQL_Server.
 10. “Oracle Database.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Oracle_Database.
 11. “MySQL.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/MySQL>.
 12. Bjarne Stroustrup, “C++ Applications.” Accessed March 18. 2023. <https://www.stroustrup.com/applications.html>.

In addition, we'll introduce Docker and show how to use `g++` and `clang++` in Docker containers that can execute on Windows, macOS or Linux. Docker is particularly important for this book's macOS users. Apple's Xcode environment does not yet support several key C++20 features we use throughout the book. Docker will enable macOS users (and Windows and Linux users) to compile C++ programs with the latest `g++` and `clang++` versions. You may want to read only the test-drive(s) required for your course or projects in industry.

This Book's Architecture

Before you "dig in," we recommend getting a "40,000-foot" view of the book's architecture to understand where you're headed as you prepare to learn the powerful language that is C++20. To do so, we recommend reviewing the following items:

- In the Preface, the one-page, full-color Table of Contents diagram presents a high-level overview of the book's architecture. You can view a scalable PDF version of this diagram at
<https://deitel.com/cpphtp11>
- The preceding website also contains a concise introduction to the book, a bullet list of its key features and testimonial comments from university professors and C++ subject-matter experts who reviewed the prepublication manuscript.
- The Preface presents the "soul of the book" and our approach to Modern C++ programming. We introduce the early chapters' "objects-natural approach," in which you'll use small numbers of simple C++ statements to make powerful existing classes perform significant tasks—long before you learn how to create your own custom classes in Chapter 9. Be sure to read the Tour of the Book, which points out the key features of each chapter and enumerates the book's 50 more substantial case studies and case-study exercises. As you read the Tour, you might also want to refer to the Table of Contents diagram.

1.2 Hardware

Computers process data under the control of instructions called **programs**. These guide the computer through ordered actions specified by people called computer **programmers**.

A computer's hardware consists of various physical devices, such as the keyboard, screen, mouse, solid-state disks, memory and processing units. Computing costs are dropping dramatically due to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon computer chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it's an ingredient in ordinary sand. Silicon-chip technology has made computing so economical that computers and computerized devices have become commodities.

1.2.1 Computer Organization

Regardless of physical differences, computers have various **logical units** or sections.

Input Unit

This “receiving” section obtains information (data and computer programs) from **input devices** and places it at the other units’ disposal for processing. Computers receive most user input through keyboards, touch screens, mice and touchpads. Other forms of input include:

- receiving voice commands,
- scanning images, barcodes or QR codes,
- reading data from secondary storage devices (such as solid-state drives, Blu-ray Disc™ drives and USB flash drives—also called “thumb drives” or “memory sticks”),
- receiving video from a webcam,
- receiving information from the Internet (such as when you stream videos from YouTube® or download e-books from Amazon),
- receiving position data from a GPS device,
- receiving motion and orientation information from an accelerometer (a device that responds to up/down, left/right and forward/backward acceleration) in a smartphone or wireless game controllers, such as those for Microsoft® Xbox®, Nintendo Switch® and Sony® PlayStation®, and
- receiving voice input from intelligent assistants like Apple® Siri®, Amazon® Alexa®, Microsoft® Cortana® and Google Home™.

Output Unit

This “shipping” section takes the information the computer has processed and places it on various **output devices** to make it available outside the computer. Most information that’s output from computers today is

- displayed on screens,
- printed on paper (“going green” discourages this),
- printed on 3D printers,
- played as audio or video on smartphones, tablets, PCs and giant screens in sports stadiums,
- transmitted over the Internet, or
- used to control other devices, such as self-driving cars, robots and “intelligent” appliances.

Information is also commonly output to secondary storage devices, such as solid-state drives (SSDs), hard drives, USB flash drives and DVD drives. Popular recent forms of output are smartphone and game-controller vibration, virtual reality devices like Meta Quest® 2, Meta Quest® Pro, Sony® PlayStation® VR and Samsung Gear VR®, and mixed reality devices like Magic Leap® 2 and Microsoft HoloLens® 2.

Memory Unit

This rapid-access, relatively low-capacity “warehouse” section retains information entered through the input unit, making it immediately available for processing when needed. The

memory unit also retains processed information until the output unit can place it on output devices. Information in the memory unit is **volatile**—it's typically lost when the computer's power is turned off. The memory unit is often called either **memory**, **primary memory** or **RAM** (Random Access Memory). Main memories on desktop and notebook computers contain as much as 128 GB of RAM, though 8 to 32 GB is most common. GB stands for gigabytes; a **gigabyte** is approximately one billion bytes. A **byte** is eight bits. A **bit** (short for “*binary digit*”) is either a 0 or a 1.

Arithmetic and Logic Unit (ALU)

This “manufacturing” section performs calculations (e.g., addition, subtraction, multiplication and division) and makes decisions (e.g., comparing two items from the memory unit to determine whether they're equal). In today's systems, the ALU is part of the next logical unit, the CPU.

Central Processing Unit (CPU)

This “administrative” section coordinates and supervises the operation of the other sections. The CPU tells

- the input unit when to read information into the memory unit,
- the ALU when to use information from the memory unit in calculations, and
- the output unit when to send information from the memory unit to specific output devices.

Most computers today have **multi-core processors** that economically implement multiple processors on a single integrated circuit chip. Such processors can perform many operations simultaneously. We say more about these in Section 1.2.2.

Secondary Storage Unit

This is the long-term, high-capacity “warehousing” section. Programs and data not actively being used by the other units are placed on secondary storage devices until they're again needed, possibly hours, days, months or even years later. Information on secondary storage devices is **persistent**—it's preserved even when the computer's power is turned off. Secondary storage information takes much longer to access than information in primary memory, but its cost is much less than RAM. Examples of secondary storage devices include solid-state drives (SSDs), USB flash drives and read/write Blu-ray drives. Many current drives hold terabytes (TB) of data—each **terabyte** is approximately one trillion bytes. Typical desktop and notebook-computer secondary storage devices hold up to 8 TB, and some recent desktop-computer hard drives hold up to 26 TB.¹³ The largest commercial SSD holds up to 100 TB (and costs about \$40,000).¹⁴

13. “History of hard disk drives.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/History_of_hard_disk_drives.

14. Desire Athow. “Largest SSDs and hard drives of 2023.” Accessed March 18, 2023. <https://www.techradar.com/best/large-hard-drives-and-ssds>.



Checkpoint

1 (*True/False*) Information in the memory unit is persistent—it's preserved even when the computer's power is turned off

Answer: False. Information in the memory unit is volatile—it's typically lost when the computer's power is turned off.

2 (*Fill-In*) Most computers today have _____ processors that implement multiple processors on a single integrated-circuit chip. Such processors can perform many operations simultaneously.

Answer: multi-core.

1.2.2 Moore's Law, Multi-Core Processors and Concurrent Programming

Many of today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. **Supercomputers** already perform over one million trillion (a quintillion) instructions per second! As of November 2022, the USA's Frontier (from Hewlett Packard Enterprise) is the world's fastest supercomputer¹⁵—it can perform 1.102 quintillion calculations per second (1.102 exaflops)¹⁶. For perspective, this supercomputer can perform in one second almost 140 million calculations for every person on the planet!¹⁷ And supercomputing “upper limits” are growing quickly.

Moore's Law

You probably expect to pay at least a little more every year for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. Over the years, hardware costs have fallen rapidly.

For decades, computer processing power approximately doubled inexpensively every couple of years. This remarkable trend is called **Moore's law**, named for Gordon Moore, co-founder of Intel and the person who identified the trend in the 1960s. Intel is a leading manufacturer of processors in today's computers and embedded systems, such as smart home appliances, home security systems, robots, intelligent traffic intersections and more. Moore's law and related observations apply especially to:

- the amount of memory that computers have for programs and data,
- the amount of secondary storage they have to hold programs and data, and
- their processor speeds—that is, the speeds at which computers execute programs to do their work.

Key executives at computer-processor companies NVIDIA and ARM have indicated that Moore's Law no longer applies. However, Intel's CEO says it is “alive and well”—



15. “Top 500.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/TOP500#TOP_500.

16. “Flops.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/FLOPS>.

17. For perspective on how far computing performance has come, consider this: In his early computing days in the 1960s, one of the authors, Harvey Deitel, used the Digital Equipment Corporation PDP-1 (<https://en.wikipedia.org/wiki/PDP-1>), which was capable of performing only 93,458 operations per second, and the IBM 1401 (<http://www.ibm-1401.info/1401GuidePosterV9.html>), which performed only 86,957 operations per second.

they are working on manufacturing advances that will enable more transistors on each chip.^{18,19} Computer processing power continues to increase but relies on new processor designs, such as multi-core processors (Section 1.2.1).

Multi-Core Processors and Performance

Most computers today have multi-core processors that economically implement multiple processors on a single integrated circuit chip. A **dual-core processor** has two CPUs, a **quad-core processor** has four, and an **octa-core processor** has eight. Intel has some processors with up to 72 cores. Our primary testing computer uses an eight-core Intel processor. Apple's recent M2 Pro and M2 Max processors have 12-core CPUs. In addition, the top-of-the-line M2 Pro has a 19-core graphics processing unit (GPU), while the top-of-the-line M2 Max processor has a 28-core GPU. Both have a 16-core "neural engine" for machine learning.²⁰ Intel is working on processors with up to 80.²¹ AMD is working on processors with 192 and 256 cores.²² The number of cores will continue to grow. Today's most powerful GPUs used in high-end gaming computers and artificial-intelligence applications often have thousands of cores in their GPUs. For example, at the time of this writing, the forthcoming NVIDIA RTX 4090 Ti is expected to have 18,176 cores²³



In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, enabling your program to complete faster. Taking full advantage of multi-core architecture requires writing multithreaded applications (Chapter 17). When a program splits tasks into separate threads, a multi-core system can run those threads in parallel when a sufficient number of cores is available.

Interest in multithreading is rising quickly because of the proliferation of multi-core systems. Standard C++ multithreading was one of the most significant updates introduced in C++11. Each subsequent C++ standard has added higher-level capabilities to simplify multithreaded application development. Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, discusses creating and managing multithreaded C++ applications. Chapter 18 introduces C++20 coroutines, which enable concurrent programming with a simple sequential-like coding style.



-
18. Esther Shein. "Moore's Law turns 55: Is it still relevant?" April 17, 2020. Accessed March 18, 2023. <https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant>.
 19. Leswing, Kif. "Intel says Moore's Law is still alive and well. Nvidia says it's ended." September 27, 2022. Accessed March 18, 2023. <https://www.cnbc.com/2022/09/27/intel-says-moores-law-is-still-alive-nvidia-says-its-ended.html>.
 20. "Apple unveils M2 Pro and M2 Max: next-generation chips for next-level workflows," January 23, 2023. Accessed March 18, 2023. <https://www.apple.com/newsroom/2023/01/apple-unveils-m2-pro-and-m2-max-next-generation-chips-for-next-level-workflows/>.
 21. Anton Shilov, "Intel's Sapphire Rapids Could Have 72-80 Cores, According to New Die Shots." April 30, 2021. Accessed March 18, 2023. <https://www.tomshardware.com/news/intel-sapphire-rapids-could-feature-80-cores>.
 22. Anthony Garreffa, "AMD EPYC 'Venice' CPU: Zen 6 with 256 cores, 512 threads... or MORE!" May 22, 2022. Accessed March 18, 2023. <https://www.tweaktown.com/news/85915/amd-epyc-venice-cpu-zen-6-with-256-cores-512-threads-or-more/index.html>.
 23. "NVIDIA GeForce RTX 4090 Ti rumored to feature 18176 cores and 24GB/24Gbps memory." Accessed March 18, 2023. <https://videocardz.com/newz/nvidia-geforce-rtx-4090-ti-rumored-to-feature-18176-cores-and-24gb-24gbps-memory>.

Computing Power Over the Years

Data is getting more massive, and so is the computing power needed for processing it. Today's processor performance is often measured in terms of **FLOPS (floating-point operations per second)**. In the early to mid-1990s, the fastest supercomputer speeds were measured in gigaflops (10^9 FLOPS). By the late 1990s, Intel produced the first teraflop (10^{12} FLOPS) supercomputers. In the early-to-mid 2000s, speeds reached hundreds of teraflops, then in 2008, IBM released the first petaflop (10^{15} FLOPS) supercomputer. As of November 2022, Hewlett Packard Enterprise's Frontier is the world's fastest supercomputer^{24,25}—it can perform 1.102 quintillion calculations per second (1.102 *exaflops*— 10^{18} FLOPS)!²⁶ Companies like IBM also are working toward exaflop supercomputers.²⁷

The **quantum computers** now under development theoretically could operate at 18,000,000,000,000,000,000 times the speed of today's "conventional computers"²⁸! This number is so extraordinary that in one second, theoretically, a quantum computer could do staggeringly more calculations than the total that have been done by all computing devices since the beginning of time. This almost unimaginable computing power could wreak havoc with blockchain-based cryptocurrencies like Bitcoin. Engineers are already rethinking blockchain²⁹ to prepare for such massive increases in computing power.³⁰

Computing power costs continue to decline, especially with advancements in processor architecture and cloud computing. People used to ask the question, "How much computing power do I need on my local system to deal with my peak processing needs?" Today, that thinking has shifted to "Can I quickly carve out in the cloud what I need temporarily for my most demanding computing workloads?" You pay for only what you use to accomplish a given task.



Checkpoint

- 1 (Fill-In)** For many decades, every year or two, computers' capacities have approximately doubled inexpensively. This remarkable trend often is called _____.

Answer: Moore's Law.

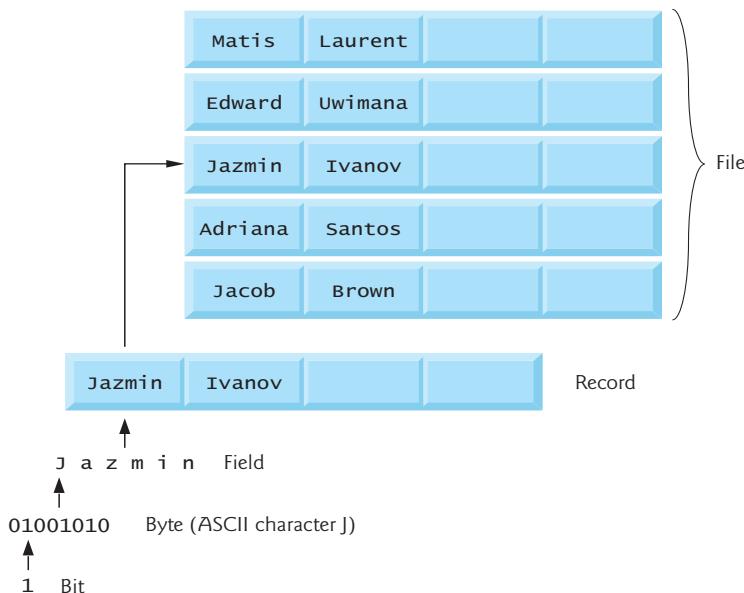
- 2 (Fill-In)** Taking full advantage of multi-core architecture requires writing _____.

Answer: multithreaded applications.

-
24. "Top 500." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/TOP500#TOP_500.
25. "Frontier (supercomputer)." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Frontier_\(supercomputer\)](https://en.wikipedia.org/wiki/Frontier_(supercomputer)).
26. "Flops." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/FLOPS>.
27. "A new supercomputing-powered weather model may ready us for Exascale." Accessed March 18, 2023. <https://www.ibm.com/blogs/research/2017/06/supercomputing-weather-model-exascale/>.
28. Norbert Biedrzycki, "Only God can count that fast — the world of quantum computing." May 12, 2017. Accessed March 18, 2023. <https://medium.com/@n.biedrzycki/only-god-can-count-that-fast-the-world-of-quantum-computing-406a0a91fcf4>.
29. "Blockchain." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Blockchain>.
30. Nathana Sharma, "Is Quantum Computing an Existential Threat to Blockchain Technology?" November 5, 2017. Accessed March 18, 2023. <https://singularityhub.com/2017/11/05/is-quantum-computing-an-existential-threat-to-blockchain-technology/>.

1.3 Data Hierarchies

Data items processed by computers can form **data hierarchies** that become larger and more complex in structure as we progress from the simplest items (called “bits”) to richer ones, such as characters and fields. The following diagram illustrates a portion of a data hierarchy:



Bits

A bit is short for “*binary digit*”—a digit that can assume one of *two* values, 0 or 1—and is a computer’s smallest data item. Remarkably, computers’ impressive functions involve only the simplest manipulations of 0s and 1s—examining a bit’s value, setting a bit’s value and reversing a bit’s value (from 1 to 0 or 0 to 1). Bits form the basis of the binary number system, which we discuss in the “Number Systems” appendix at <https://deitel.com/cpphtp11>.

Characters

Working with data in the low-level form of bits is tedious. Instead, people prefer to work with **decimal digits** (0–9), **letters** (A–Z and a–z) and **special symbols** such as

```
$ @ % & * ing ( ) - + " : ; , ? /
```

Digits, letters and special symbols are known as **characters**. The computer’s **character set** contains the characters used to write programs and represent data items. Computers process only 1s and 0s, so a computer’s character set represents each character as a pattern of 1s and 0s. C++ uses the **ASCII (American Standard Code for Information Interchange)** character set by default. C++ also supports **Unicode®** characters composed of one, two, three or four bytes (8, 16, 24 or 32 bits, respectively).³¹

31. Victor Stinner, “Programming with Unicode.” Accessed March 18, 2023. https://unicode-book.readthedocs.io/programming_languages.html.

Unicode contains characters for many of the world's languages. ASCII is a (tiny) subset of Unicode representing letters (a–z and A–Z), digits and some common special characters. You can view the ASCII subset of Unicode at

<https://www.unicode.org/charts/PDF/U0000.pdf>

For the lengthy Unicode charts for all languages, symbols, emojis and more, visit

<http://www.unicode.org/charts/>

Fields

Just as characters are composed of bits, **fields** are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters could represent a person's name, and a field consisting of decimal digits could represent a person's age in years.

Records

Several related fields can be used to compose a **record**. In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):

- Employee identification number (a whole number).
- Name (a group of characters).
- Address (a group of characters).
- Hourly pay rate (a number with a decimal point).
- Year-to-date earnings (a number with a decimal point).
- Amount of taxes withheld (a number with a decimal point).

Thus, a record is a group of related fields. All the fields listed above belong to the same employee. A company might have many employees and a payroll record for each.

Files

A **file** is a group of related records. More generally, a file contains arbitrary data in arbitrary formats. Some operating systems view a file simply as a sequence of bytes—any organization of the bytes in a file, such as into records, is a view created by the application programmer. You'll see how to do that in Chapter 8. It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information. As we'll see below, far larger file sizes are becoming increasingly common with big data.

Databases

A **database** is a collection of data organized for easy access and manipulation. The most popular model is the **relational database**, in which data is stored in simple tables of rows and columns. A table includes records and fields. For example, a table of students might include first name, last name, major, year, student ID number and grade-point-average fields. The data for each student is a record, and the individual pieces of information in each record are the fields. You can search, sort and otherwise manipulate the data based on its relationship to multiple tables or databases. For example, a university might use data from the student database combined with data from databases of courses, on-campus housing, meal plans, etc.

Big Data

The table below shows some common byte measures:

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 bytes (1024 bytes exactly)
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000) bytes
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000) bytes
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000) bytes
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000) bytes
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000) bytes
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000) bytes

The amount of data produced worldwide is enormous, and its growth is accelerating. **Big data** applications deal with massive amounts of data.

Twitter®—A Favorite Big-Data Source

One big-data source favored by developers is Twitter. As of March 2023, there are approximately 900,000,000 tweets per day.³² Though tweets are limited to 280 characters, programmers analyzing tweets can request additional metadata—such as who sent the tweet, images, videos, links, mentions of other users and more—that can total approximately 10,000 bytes per tweet. So 900,000,000 times 10,000 is about 9,000,000,000,000 bytes or 9 terabytes (TB) of data per day. That’s a nice example of big data.³³

Prediction is a challenging and often costly process, but the potential rewards for accurate predictions are remarkable. **Data mining** is the process of searching through extensive collections of data, often big data, to find insights that can be valuable to individuals and organizations. The sentiment you data-mine from tweets could help predict election results, the revenues a new movie will likely generate and the success of a company’s marketing campaign. It could also help companies spot weaknesses in competitors’ product offerings.



Checkpoint

1 *(Fill-In)* A(n) _____ is short for “binary digit”—a digit that can assume one of two values and is a computer’s smallest data item.

Answer: bit.

2 *(Fill-In)* A database is a collection of data organized for easy access and manipulation. The most popular model is the _____ database, in which data is stored in simple tables.

Answer: relational.

32. “Worldometer.” Accessed March 18, 2023. <https://www.worldometers.info/>.

33. At the time of this writing, Twitter has announced changes to their APIs and limits, so the amount of data you can access through their APIs may change.

1.4 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of such languages are in use today. These may be divided into three general types:

- Machine languages.
- Assembly languages.
- High-level languages.

Machine Languages

Any computer can directly understand only its own **machine language**, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine-dependent—a particular machine language can be used on only one type of computer. Such languages are cumbersome for humans. For example, here's a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:

```
+1300042774  
+1400593419  
+1200274027
```

In our Building Your Own Computer case study (Exercises 7.9–7.11), you'll “peel open” a “made-up” computer and look at its internal structure. We'll introduce machine-language programming, and you'll write several machine-language programs. To make this an especially valuable experience, you'll then build a software simulation of a computer on which you can execute your machine-language programs. This will give you a friendly introduction to the contemporary computer architecture concept of “virtual machines.”

Assembly Languages and Assemblers

Programming in machine language was too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of **assembly languages**. Translator programs called **assemblers** were developed to convert assembly-language programs to machine language at computer speeds. The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay:

```
load    basepay  
add     overpay  
store   grosspay
```

Although such code is clearer to humans, it's incomprehensible to computers until it's translated into machine language.

High-Level Languages and Compilers

With the advent of assembly languages, computer usage increased rapidly. However, programmers still had to use numerous instructions to accomplish even simple tasks. To speed the programming process, **high-level languages** were developed in which single statements

could accomplish substantial tasks. A typical high-level-language program contains many statements, known as the program's **source code**.

Translator programs called **compilers** convert high-level-language source code into machine language. High-level languages allow you to write instructions that look almost like everyday English and contain common mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overtimePay
```

From the programmer's standpoint, high-level languages are preferable to machine and assembly languages. C++ is among the world's most widely used high-level programming languages.

In our Building Your Own Compiler case study (Exercises 13.29–13.34), you'll build a compiler that takes programs written in a simple "made-up" high-level programming language and converts them to the Simpletron Machine Language that you learn in Exercise 7.9. Exercises 13.29–13.34 "tie" together the entire programming process. You'll write programs in a simple high-level language, compile the programs on the compiler you build, then run the programs on the Simpletron simulator virtual machine you build in Exercise 7.10.

Interpreters

Compiling large high-level language programs into machine language can take considerable computer time. **Interpreters** execute high-level language programs directly, avoiding compilation delays, but your code runs slower than compiled programs. Some programming languages, such as Java³⁴, Python³⁵ and C#³⁶, use a clever mixture of compilation and interpretation to run programs.



Checkpoint

- 1** (*Fill-In*) _____ programs, developed to execute high-level-language programs directly, avoid compilation delays, although they run slower than compiled programs
Answer: Interpreter.

- 2** (*True/False*) High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.

Answer: True.

1.5 Operating Systems

Operating systems are software that make using computers more convenient for users, software developers and system administrators. They provide services that allow applications to execute safely, efficiently and concurrently with one another. The software that contains the core operating-system components is called the **kernel**. Linux, Windows and macOS are

34. "Java virtual machine." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Java_virtual_machine#Bytecode_interpreter_and_just-in-time_compiler.

35. "The Bytecode Interpreter." Accessed March 18, 2023. <https://devguide.python.org/internals/interpreter/>.

36. "Common Language Runtime (CLR) overview." Accessed March 18, 2023. <https://learn.microsoft.com/en-us/dotnet/standard/clr>.

popular desktop/laptop operating systems—you can use any of these with this book. The most popular mobile operating systems used in smartphones and tablets are Google’s Android and Apple’s iOS.

Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS (Disk Operating System)—an enormously popular desktop/laptop operating system that users interacted with by typing commands. Windows 11 is Microsoft’s latest operating system.³⁷ Windows is a **proprietary** operating system controlled exclusively by Microsoft. It is the world’s most widely used desktop and notebook operating system, with almost 72% market share (February 2023).³⁸

Linux—An Open-Source Operating System

The **Linux operating system** is among the greatest successes of the open-source movement. Proprietary software for sale or lease dominated software’s early years. With **open source**, individuals and companies contribute to developing, maintaining and evolving the software. Anyone can then use that software for their own purposes—normally at no charge, but subject to various (typically generous) licensing requirements. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors can get removed faster, making the software more robust. Open-source increases productivity and has contributed to an explosion of innovation. You’ll use various popular open-source libraries and tools throughout this book.

Some key organizations in the open-source community are:

- **GitHub** provides tools for managing open-source software-development projects. It has millions of them underway.
- The **Apache Software Foundation** created the Apache web server and now oversees 350+ open-source projects,³⁹ including many big-data infrastructure technologies.
- The **Eclipse Foundation** oversees over 400 open-source projects, including the Eclipse Integrated Development Environment.⁴⁰
- The **Mozilla Foundation** is the creator of the Firefox web browser and several other projects, including Firefox Reality (a virtual reality web browser), Hubs (a virtual reality meeting space), Project Things (for building Internet of Things apps), WebXR Viewer (an augmented reality viewer) and Spoke (for building 3D social environments that can be visited using virtual reality devices).⁴¹

37. Alison DeNisco Rayome and Shelby Brown. “Every Difference You Should Care About Between Windows 10 and Windows 11.” November 5, 2022. Accessed March 18, 2023. <https://www.cnet.com/tech/computing/every-difference-you-should-care-about-between-windows-10-and-windows-11/>.

38. “Desktop Operating System Market Share Worldwide.” Accessed March 18, 2023. <https://gs.statcounter.com/os-market-share/desktop/worldwide>.

39. “Welcome to the Apache Projects Directory.” Accessed March 18, 2023. <https://projects.apache.org/>.

40. “Eclipse Foundation.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Eclipse_Foundation.

41. “Welcome to the Mozilla Laboratory.” Accessed March 18, 2023. <https://labs.mozilla.org/projects/>.

- **OpenML** focuses on open-source tools and data for machine learning.
- **OpenAI** researches artificial intelligence and publishes open-source tools used in AI research. They're also the creators of ChatGPT and Dall-E 2, which have each generated enormous interest in artificial intelligence. See Section 1.16 for more about these technologies.
- **OpenCV** focuses on open-source computer-vision tools that can be used across various operating systems and programming languages.
- The **Python Software Foundation** is responsible for the Python programming language.
- The **Boost** organization manages the development of the open-source Boost C++ libraries (see Section 1.7).

Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create software-based businesses now than just a decade ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.

The **Linux kernel** is the core of the most popular open-source, freely distributed, full-featured operating system. It's developed by a loosely organized team of volunteers and is popular in servers, personal computers and embedded systems (such as the computer systems at the heart of smartphones, smart TVs and automobile systems). Unlike Microsoft's Windows and Apple's macOS source code, the Linux source code is available to the public for examination and modification, and is free to download and install. As a result, Linux users benefit from a massive community of developers actively debugging and improving the kernel, and the ability to customize the operating system to meet specific needs.

Apple's macOS and Apple's iOS for iPhone® and iPad® Devices

Apple, founded in 1976 by Steve Jobs and Steve Wozniak, quickly became a leader in personal computing. In 1979, Jobs and several Apple employees visited Xerox PARC (Palo Alto Research Center) to learn about Xerox's desktop computer that featured a graphical user interface (GUI). That GUI served as the inspiration for the Apple Macintosh, launched in 1984.

The Objective-C programming language, created by Stepstone in the early 1980s, added object-oriented programming (OOP) capabilities to the C programming language. Steve Jobs left Apple in 1985 and founded NeXT Inc. In 1988, NeXT licensed Objective-C from Stepstone. NeXT developed an Objective-C compiler and libraries, which were used as the platform for the NeXTSTEP operating system's user interface and Interface Builder (for constructing graphical user interfaces).

Jobs returned to Apple in 1996 when they bought NeXT. Apple's **macOS operating system** is a descendant of NeXTSTEP. Apple has several other proprietary operating systems derived from macOS:

- **iOS** is used in iPhones.
- **iPadOS** is used in iPads.
- **watchOS** is used in Apple Watches.
- **tvOS** is used in Apple TV devices.

In 2014, Apple introduced its Swift programming language, which it open-sourced in 2015. The Apple app-development community has largely shifted from Objective-C to Swift.

Google's Android

Android—the most widely used mobile and smartphone operating system—is based on the Linux kernel, the Java programming language and now, the open-source Kotlin programming language. Android is open-source and free. Though you can't develop Android apps purely in C++, you can incorporate C++ code.⁴²

According to [statista.com](https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/), in the fourth quarter of 2022, 71.8% of smartphones use Android and 27.6% use Apple iOS.⁴³ This represents an increasing market for Apple iOS. The Android operating system is used in numerous smartphones, e-reader devices, tablets, TVs, in-store touch-screen kiosks, cars, robots, multimedia players and more.

Billions of Computerized Devices

Billions of personal computers and an even larger number of mobile devices are now in use. The explosive growth of smartphones, tablets and other devices has created significant opportunities for mobile app developers. The following table lists many computerized devices, each of which can be part of the Internet of Things (see Section 1.12).

Some computerized devices		
Activity trackers— Apple Watch®, FitBit®, ...	Personal assistants— Amazon Echo® (Alexa®), Apple HomePod® (Siri®), Google Home™ (Google Assistant™)	Smart home and building controls—lights, video cameras, doorbells, irriga- tion controllers, security devices, smart locks, smart plugs, smoke detectors, thermostats, air vents
ATMs	Personal computers	Smart meters
e-Readers	Point-of-sale terminals	Smartcards (credit cards)
Gaming consoles	Printers	Smartphones
GPS navigation systems	Robots	Thermostats
Home appliances	Self-driving cars	Tracking devices
Medical devices— blood glucose monitors, blood pressure monitors, CT scanners, electrocardio- grams (EKG/ECG), electro- encephalograms (EEG), heart monitors, MRIs, pace- makers, ...	Sensors— chemical, earthquake, gas, GPS, humidity, light, opti- cal, motion, pressure, tem- perature, tsunami, ...	Transportation passes
Parking meters	Server computers	TVs and TV set-top boxes
		Vehicles
		Wireless network devices



Checkpoint

I (Fill-In) Windows is a(n) _____ operating system—it's controlled by Microsoft exclusively.

Answer: proprietary.

-
42. "Add C and C++ code to your project." Accessed March 18, 2023. <https://developer.android.com/studio/projects/add-native-code>.
 43. "Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022." Accessed March 10, 2023. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009.

2 (*True/False*) Proprietary code is often scrutinized by a much larger audience than open-source software, so errors often get removed faster.

Answer: False. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster.

1.6 C and C++

C Programming Language

C was implemented in 1972 by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system’s development language. Today, most of the code for general-purpose operating systems is written in C or C++. C is available for most computers and is hardware independent. With careful design, it’s possible to write C programs that are portable to most computers.

The widespread use of C with various kinds of computers (sometimes called hardware platforms) led to many variations. The American National Standards Institute (ANSI) cooperated with the International Organization for Standardization (ISO) to standardize C worldwide—the joint standard document was published in 1990. An updated standard is currently under development.

C++ Programming Language

C++⁴⁴ was developed by Bjarne Stroustrup in 1979 at Bell Laboratories as an extension of C. Originally called “C with Classes,” it was renamed C++ in the early 1980s—the name derives from C’s increment operator, ++, which adds 1 to a number. It provides many features that “spruce up” C, but more importantly, it includes capabilities for object-oriented programming inspired by the Simula simulation programming language. As you’ll see, C++ has evolved over the last 40 years to enable programmers to use five different programming models, each of which you’ll learn in this book:

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

C++ is the premiere language for building high-performance, business-critical and mission-critical computing systems, including operating systems, real-time systems, embedded systems, game systems, banking systems, air-traffic-control systems, communications systems and more. Even for systems programmed primarily in other languages, C++ is often used to build the performance-intensive portions of those systems. Stroustrup maintains a lengthy list of C++ applications on his website at

<https://www.stroustrup.com/applications.html>

44. Stroustrup, Bjarne. “A History of C++: 1979–1991.” Accessed March 18, 2023. <https://www.stroustrup.com/hop12.pdf>.

Modern C++

Today's C++ programmers build systems and system components using Modern C++, which includes the C++20, C++17, C++14 and C++11 standards—the digits in each name indicate the year in the 2000s when that version was approved by the ISO C++ standard committee. Many of the language and library features introduced by these standards make C++ easier to use, improve runtime performance and software quality, and enable applications to take advantage of today's multi-core systems.

C++ is over four decades old, so there is much existing C++ code using older programming idioms—this is typically referred to as **legacy code**. We emphasize Modern C++ idioms in this book.

C++ Core Guidelines

Bjarne Stroustrup, Herb Sutter (the ISO C++ committee chair) and several other contributors⁴⁵ have created the evolving **C++ Core Guidelines**,⁴⁶ containing extensive recommendations and sample code to help you use Modern C++ correctly and effectively. Various **code-analysis tools** (discussed in Preface Section 7) enable you to check your code to ensure it follows these recommendations. We cite the C++ Core Guidelines frequently throughout this book.



Checkpoint

1 (*Fill-In*) Most code for general-purpose operating systems is written in _____.

Answer: C or C++.

2 (*Fill-In*) The _____ provide recommendations and sample code to help you use Modern C++ correctly and effectively.

Answer: C++ Core Guidelines.

1.7 C++ Standard Library and Open-Source Libraries

C++ programs consist of pieces called **functions** and **classes**. You can program all the pieces you'll need yourself. Instead, most C++ programmers reuse the rich collections of classes and functions in the **C++ standard library**. Thus, there are really two parts to learning C++ programming:

- learning the C++ language itself (often referred to as the “core language”), and
- learning how to use the C++ standard library’s classes and functions.

The Boost Project and Other Open Source Libraries

There are enormous numbers of open-source C++ libraries that can help you perform significant tasks by writing only modest amounts of code. The **Boost C++ libraries** provide 168 powerful open-source libraries.⁴⁷ Boost serves as a “breeding ground” for new capa-

45. “C++ Core Guidelines—Acknowledgments.” Accessed March 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-ack>.

46. “C++ Core Guidelines.” Accessed March 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>.

47. “Boost 1.81.0 Library Documentation.” Accessed March 18, 2023. https://www.boost.org/doc/libs/1_81_0/.

bilities that might eventually be incorporated into the C++ standard libraries. The following StackOverflow post lists Modern C++ libraries and language features that evolved from the Boost libraries:⁴⁸

<https://stackoverflow.com/a/8852421>

We use the **Boost Multiprecision library** in our objects-natural case studies on super-sized integers (Section 3.14) and precise monetary calculations (Section 4.14).

On GitHub, developers contribute to almost 56,000 C++ code repositories:

<https://github.com/topics/cpp>

In addition, pages such as

<https://github.com/fffaraz/awesome-cpp>

<https://github.com/rigtorp/awesome-modern-cpp>

provide curated lists of popular C++ libraries for many application areas—you'll enjoy exploring these sites.

Building-Block Approach to C++ Development

Throughout this book, we encourage a **building-block approach** to creating programs. When programming in C++, you'll typically use the following building blocks:

- C++ standard library classes and functions,
- open-source C++ library classes and functions,
- functions and classes you create (after studying Chapters 5 and 9, respectively), and
- classes and functions other people have created and made available to you.

The advantage of creating your own classes and functions is knowing exactly how they work. The disadvantage is the time-consuming effort that goes into designing, developing, debugging and performance-tuning them. Throughout the book, we focus on using the existing C++ standard library and various open-source libraries to leverage your program-development efforts and avoid “reinventing the wheel.” This is called **software reuse** and is a key part of our objects-natural teaching methodology.



Using C++ Standard Library classes and functions instead of writing your own versions can improve program performance because they're written carefully to perform efficiently. This technique also shortens program development time. Using C++ Standard Library classes and functions instead of writing your own also improves program portability because they're included in every C++ implementation.



Checkpoint

1 *(Fill-In)* Most C++ programmers take advantage of the rich collection of existing functions called the _____.

Answer: C++ standard library.

2 *(Fill-In)* Avoid “reinventing the wheel” Instead, use existing pieces from code libraries. This is an example of _____.

Answer: software reuse.

48. Kennytm, Answer to “Which Boost Features Overlap with C++11?” Accessed March 18, 2023.
<https://stackoverflow.com/a/8852421>.

1.8 Other Popular Programming Languages

The following is a brief intro to several other popular programming languages:

- **Python** is an open-source, object-oriented language that was released publicly in 1991. It was developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam. Python has rapidly become one of the world's most popular programming languages, especially for scientific computing, data science and artificial intelligence. It has extensive standard libraries and thousands of third-party open-source libraries.⁴⁹
- **Java**⁵⁰—a C++-based object-oriented programming language—was created in 1991 when Sun Microsystems funded an internal corporate research project led by James Gosling. A key Java goal is “write once, run anywhere,” enabling developers to write programs that run on a wide variety of computer systems. Java is used in many kinds of applications, including big data (Section 1.15), the Internet of Things (Section 1.12; devices connected to the Internet, such as sensors, smartphones, tablets, television set-top boxes, appliances and automobiles), enterprise applications, web servers (the computers that provide the content to our web browsers), video games and more.⁵¹ Originally, Java was the official Android app-development language, though over 60% of Android app developers now use Kotlin.⁵²
- **C#**⁵³ (based on C++ and Java) is one of Microsoft's three primary object-oriented programming languages—the other two are Visual C++ and Visual Basic. C# was developed to integrate the web into computer applications and is now widely used to develop many kinds of applications. It is Microsoft's primary language for Metaverse (Section 1.13) development.⁵⁴ Microsoft now offers open-source versions of C# and Visual Basic.
- **Visual Basic**⁵⁵ was originally based on the **BASIC** programming language, developed in the 1960s at Dartmouth College to familiarize novices with programming. Both C# and Visual Basic use the vast code libraries of Microsoft's .NET platform. Components written in both languages typically can be used interchangeably, enabling developers to work with their preferred programming language to build powerful .NET-based applications.

-
49. “Python programming language.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)).
 50. “Java programming language.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language)).
 51. “What is Java used for?” March 14, 2022. Accessed March 18, 2023. <https://www.future-learn.com/info/blog/what-is-java-used-for>.
 52. “Build Better Apps with Kotlin.” Accessed March 18, 2023. <https://developer.android.com/kotlin/build-better-apps>.
 53. “C Sharp (programming language).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)).
 54. Mahesh Chand, “What is Microsoft Mesh?” January 5, 2022. Accessed March 18, 2023. <https://www.c-sharpcorner.com/article/what-is-microsoft-mesh/>.
 55. “Visual Basic (.NET).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Visual_Basic_\(.NET\)](https://en.wikipedia.org/wiki/Visual_Basic_(.NET)).

- JavaScript⁵⁶ is widely used to add programmability to web pages (e.g., animations, user interactivity and more). All major web browsers support it. Many visualization libraries in other programming languages output JavaScript to create interactive visualizations you can view in your web browser. Tools such as NodeJS also enable JavaScript to run outside of web browsers. A key aspect of Modern JavaScript development is creating powerful web applications using JavaScript frameworks such as React, Angular, Vue and others.^{57,58}
- Swift⁵⁹ was introduced by Apple in 2014 and is their programming language for developing iOS and macOS apps. Swift is a contemporary language with popular features from Objective-C, Java, C#, Ruby, Python and other languages. Swift is open-source, so it also can be used on non-Apple platforms.
- Go⁶⁰ is a high-performance, open-source systems programming language introduced by Google in 2009. It was designed specifically for developing systems to take advantage of today's multi-core, networked systems. The language is used widely at Google and in many popular open-source applications.
- Rust⁶¹ is a high-performance, open-source, general-purpose programming language that began as a personal project of Mozilla research developer Graydon Hoare in 2006, then became an official Mozilla project released in 2015. A key language feature is its support for concurrency. Some companies using Rust include Amazon, Google, Microsoft, Facebook and Dropbox.



Checkpoint

1 (*Fill-In*) Today, most code for general-purpose operating systems and other performance-critical systems is written in _____.

Answer: C or C++.

2 (*Fill-In*) A key goal of the _____ programming language is “write once, run anywhere,” enabling developers to write programs that will run on a great variety of computer systems and computer-controlled devices.

Answer: Java.

-
56. “JavaScript.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/JavaScript>.
57. Eric Elliott, “Top JavaScript Frameworks and Technology 2023.” February 19, 2023. Accessed March 18, 2023. <https://medium.com/javascript-scene/top-javascript-frameworks-and-technology-2023-4e4a06d6be93>.
58. Simran Kaur Arora, “10 Best JavaScript Frameworks to Use in 2023.” December 29, 2022. Accessed March 18, 2023. <https://hackr.io/blog/best-javascript-frameworks>.
59. “Swift (programming language).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Swift_\(programming_language\)](https://en.wikipedia.org/wiki/Swift_(programming_language)).
60. “Go (programming language).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Go_\(programming_language\)](https://en.wikipedia.org/wiki/Go_(programming_language)).
61. “Go (programming language).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Go_\(programming_language\)](https://en.wikipedia.org/wiki/Go_(programming_language)).

1.9 Introduction to Object Orientation

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in Chapter 9—the **classes** objects come from, are essentially **reusable** software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any noun can be reasonably represented as a software object in terms of **attributes** (e.g., name, color and size) and **behaviors** (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

Automobile as an Object

Let's begin with a simple analogy. Suppose you want to drive a car and make it go faster by pressing its accelerator pedal. What must happen before you can do this? Well, someone has to **design** the car before you can drive it. A car typically begins as engineering drawings, similar to the **blueprints** that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal **hides** from the driver the complex mechanisms that make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel hides the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be **built** from the engineering drawings that describe it. A completed car has an **actual** accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must **press** the pedal to accelerate the car.

Functions, Member Functions and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a function. The function houses the program statements that perform its task. It **hides** these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In C++, we often create a program unit called a **class** to house the set of functions that perform the class's tasks—these are known as the class's **member functions**. For example, a class representing a bank account might contain a member function to **deposit** money to an account, another to **withdraw** money from an account and a third to **query** the account's current balance. A class is similar to a car's engineering drawings, which house the design of an accelerator pedal, brake pedal, steering wheel, and so on.

Instantiation

Just as someone has to **build a car** from its engineering drawings before you can drive a car, you must **build an object** from a class before a program can perform the tasks defined by the class's member functions. The process of doing this is called **instantiation**, and an object is then referred to as an **instance** of its class.

Reuse

Just as a car's engineering drawings can be **reused** many times to build many cars, you can **reuse** a class many times to build many objects. Reusing existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems because existing classes and components often have been extensively **tested, debugged and performance-tuned**. Just as the notion of **interchangeable parts** was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution spurred by object technology.



Messages and Member-Function Calls

When you drive a car, pressing its gas pedal sends a **message** to the car to perform a task—that is, to “go faster.” Similarly, you **send messages to an object**. Each message is implemented as a **member-function call** that tells a member function of the object to perform its task. For example, a program might call a particular bank account object’s **deposit** member function to increase the account’s balance by the deposit amount.

Attributes and Data Members

Besides having capabilities to accomplish tasks, a car also has **attributes**, such as its color, number of doors, amount of gas in its tank, current speed and record of total miles driven (i.e., its odometer reading). Like its capabilities, the car’s attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive a car, these attributes are “carried along” with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank but not how much is in the tanks of other cars.

An object, similarly, has attributes that it carries along as it’s used in a program. These attributes are specified as part of the object’s class. For example, a bank-account object has a **balance attribute** representing the amount of money in the account. Each bank-account object knows the balance in the account it represents but not the balances of the other accounts in the bank. Attributes are specified by the class’s **data members**.

Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and member functions into objects created from those classes—an object’s attributes and member functions are intimately related. Objects may communicate with one another, but they’re normally not allowed to know how other objects are implemented internally. Those details are **hidden** inside the objects. As we’ll see, this **information hiding** is crucial to good software engineering.



Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**. The new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of the class “convertible” certainly **is an** object of the more **general** class “automobile,” but more **specifically**, the roof can be raised or lowered.

Object-Oriented Analysis and Design

Soon you’ll be writing programs in C++. How will you create the **code** for your programs? Perhaps, like many programmers, you’ll turn on your computer and start typing. This

approach may work for small programs (like the ones we present in the book's early chapters), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining **what** the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding **how** the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. C++ has **object-oriented programming (OOP)** capabilities that enable you to implement object-oriented designs as working systems.



Checkpoint

1 (*Fill-In*) The size, shape, color and weight of an object are _____ of the object's class.

Answer: attributes.

2 (*True/False*) Objects, or more precisely, the classes objects come from, are essentially reusable software components.

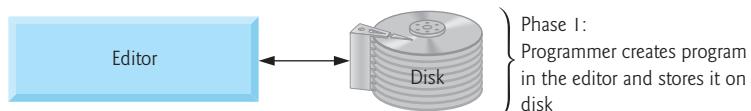
Answer: True.

I.10 Simplified View of a C++ Development Environment

C++ systems generally consist of three parts: a program development environment, the language and the C++ Standard Library. C++ programs typically go through six phases: edit, preprocess, compile, link, load and execute. The following discussion explains a typical C++ program development environment.

Phase 1: Editing a C++ Program

Phase 1 consists of editing a file with an editor program, normally known simply as an editor:

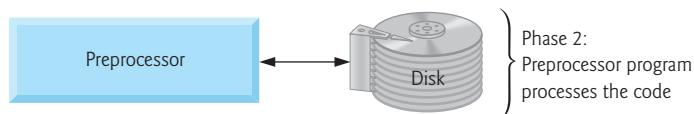


You type a C++ program's **source code** using the editor, make any necessary corrections and save the program on your computer's disk. C++ source code filenames often end with the .cpp, .cxx, .cc or .C (uppercase) extensions, which indicate that a file contains C++ source code. See the documentation for your C++ compiler for more information on filename extensions. Two editors widely used on Linux systems are vim and emacs. You also can use a simple text editor, such as TextEdit on macOS, Notepad on Windows or Microsoft's Visual Studio Code (a cross-platform, code-oriented text editor with plug-ins supporting many languages).

Integrated development environments (IDEs) are available from many major software suppliers. IDEs provide tools that support the software development process, including editors for writing and editing programs and debuggers for locating **logic errors**—errors that cause programs to execute incorrectly. Some popular IDEs include Microsoft® Visual Studio Community Edition (a complete IDE), Eclipse, JetBrains CLion®, Apple’s Xcode® and CodeLite.

Phase 2: Preprocessing a C++ Program

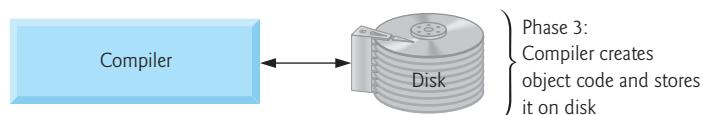
In Phase 2, you give the command to **compile** the program:



In a C++ system, a **preprocessor** program executes automatically before the compiler’s translation phase begins (so we call preprocessing Phase 2 and compiling Phase 3). The C++ preprocessor obeys commands called **preprocessing directives**, which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually include (i.e., copy into the program file) other text files to be compiled and perform various text replacements. The most common preprocessing directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in the “Preprocessor” appendix at <https://deitel.com/cpphtp11>.

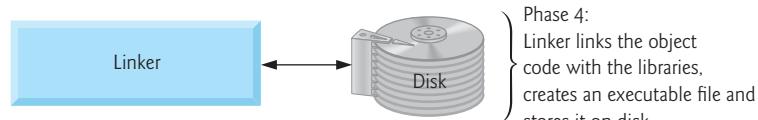
Phase 3: Compiling a C++ Program

In Phase 3, the compiler translates the C++ program into machine-language code—also referred to as **object code**:



Phase 4: Linking to Create an Executable Program

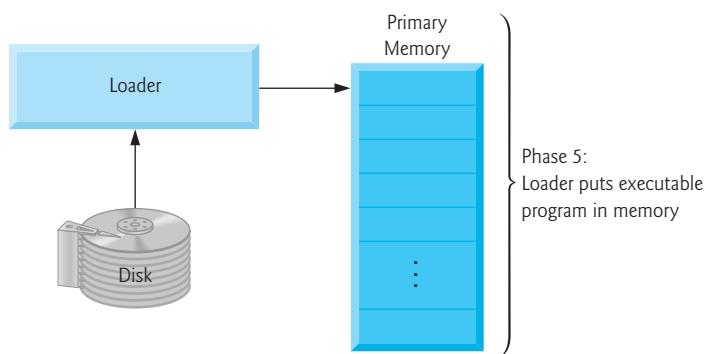
Phase 4 is called **linking**. C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project:



The object code produced by the C++ compiler typically contains “holes” due to these missing parts. A **linker** links the object code with the code for the missing functions to produce an **executable program** (with no missing pieces).

Phase 5: Loading an Executable Program

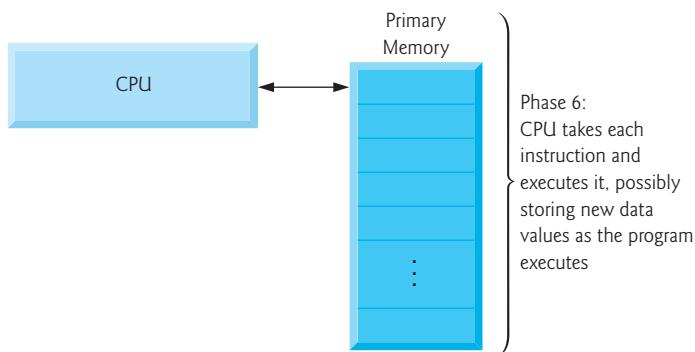
Phase 5 is called **loading**. Before a program can be executed, it must first be placed in memory:



This is done by the **loader**, which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.

Phase 6: Executing the Program

Finally, the computer, under the control of its CPU, **executes** the program one instruction at a time:



Modern multi-core computer architectures can execute several instructions in parallel.

Problems That May Occur at Execution Time

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss throughout this book. For example, an executing program might try to divide by zero (an illegal operation for integer arithmetic in C++). This would cause the C++ program to display an error message. If this occurred, you'd have to return to the editing phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections fixed the problem(s). Certain C++ functions take their input from `cin` (the **standard input stream**; pronounced “see-in”), which is normally the keyboard, but `cin` can be redirected to another device. Data is often output to `cout` (the **standard output stream**; pronounced “see-out”), which is normally the computer screen, but `cout` can be redirected to another device. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to other devices, such as disks, hardcopy printers, or even transmitted over the Inter-

net. There is also a **standard error stream** referred to as **`cerr`**. The `cerr` stream (normally connected to the screen) displays error messages.

Err Errors such as division by zero occur as a program runs, so they're called **runtime errors or execution-time errors**. **Fatal runtime errors** cause programs to terminate immediately without having successfully performed their jobs. **Nonfatal runtime errors** allow programs to run to completion, often producing incorrect results.



Checkpoint

1 *(Fill-In)* C++ programs typically go through six phases to be executed: _____, _____, _____, _____, _____ and _____.

Answer: edit, preprocess, compile, link, load, execute.

2 *(Fill-In)* A(n) _____ occurs when the compiler cannot recognize a statement because it violates the rules of the language, and an error that occurs as a program runs is called a(n) _____ or execution-time error.

Answer: syntax error, runtime error.

1.11 Test-Driving a C++20 Application Various Ways

Please read the Before You Begin section before reading this section and attempting any of the test-drives. In this section, you'll compile, run and interact with your first C++ application⁶²—a guess-the-number game, which picks a random number from 1 to 1,000 and prompts you to guess it. If you guess the randomly selected number, 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. There's some nice computer science behind this game—in a later chapter, you'll explore the **binary search technique**. You'll program the guess-the-number game in Exercise 5.22.

Summary of the Compiler and IDE Test-Drives

We'll show how to compile and execute C++ code using:

- Microsoft Visual Studio Community edition for Windows (Section 1.11.1),⁶³
- GNU `g++` in a shell on Linux (Section 1.11.2),
- `g++` in a shell running inside the GNU Compiler Collection (GCC) Docker container (Section 1.11.3), and
- LLVM `clang++` (the command-line version of the Clang C++ compiler) in a shell running inside a Docker container (Section 1.11.4).
- Apple Xcode on macOS (Section 1.11.5),

You can read only the section that corresponds to your platform. To use the Docker containers for `g++` and `clang++`, you must have Docker installed and running, as discussed in the Before You Begin section that follows the Preface.

62. We intentionally do not cover the code for this C++ program here. Its purpose is simply to demonstrate compiling and running a program using each compiler we discuss in this section. We present secure random-number generation in Chapter 5.

63. If you're interested in using Microsoft's Visual Studio Code cross-platform code editor to program in C++, see the instructions at <https://code.visualstudio.com/docs/languages/cpp>.

1.11.1 Compiling and Running on Windows with Visual Studio Community Edition

In this section, you'll run a C++ program on Windows using Microsoft Visual Studio Community edition. There are several versions of Visual Studio available—on some versions, the options, menus and instructions we present might differ slightly.

Step 1: Checking Your Setup

If you have not already done so, read the **Before You Begin** section to install the IDE and download the book's code examples.

Step 2: Launching Visual Studio

Open Visual Studio from the **Start** menu. Dismiss this initial Visual Studio window by pressing the *Esc* key. Do not click the X in the upper-right corner—that will terminate Visual Studio. You can access this window at any time by selecting **File > Start Window**. We use > to indicate selecting a menu item from a menu, so **File > Open** means “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**. A solution contains one or more projects. Each application in this book requires only a single-project solution. Multi-project solutions are used to create large-scale applications. For our code examples, you'll begin with an **Empty Project** and add files to it.



When you first open Visual Studio, a dialog will appear containing links to your recently opened solutions in the **Open Recent** column and several shortcuts in the **Get Started** column. For this test drive, we'll create a new project directly in Visual Studio, so press the *Esc* key to dismiss this initial dialog. To create a project:

1. Select **File > New > Project...** to display the **Create a New Project** dialog.
2. Depending on your Visual Studio version and its installed features, many project types may be displayed. You can filter your choices using the **Search for templates** textbox and the drop-down lists below it. In the drop-down list that initially says **All languages**, select **C++**.
3. Select the **Empty Project** template with the tags **C++**, **Windows** and **Console**. This project template is for programs that execute in a **Command Prompt** window—this is a text-based window in which programs can display text and receive text typed by the user at the keyboard. Click **Next** to display the **Configure your new project** dialog.
4. Provide a **Project name** and **Location**. For the **Project name**, we specified `cpp20_test`. For the **Location**, we selected this book's `examples` folder. Click **Create** to place your project into the following folder (or the folder you specified) and open your new project in Visual Studio:

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

When you edit C++ code, Visual Studio displays each file as a separate tab within the window. The **Solution Explorer**—docked to Visual Studio's left or right side—is for view-

ing 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 to the Project

Next, you'll add to the project you created in *Step 3* the file `GuessNumber.cpp`, containing the C++ code that implements the guess-the-number game. 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, select `GuessNumber.cpp` and click **Add**.⁶⁴

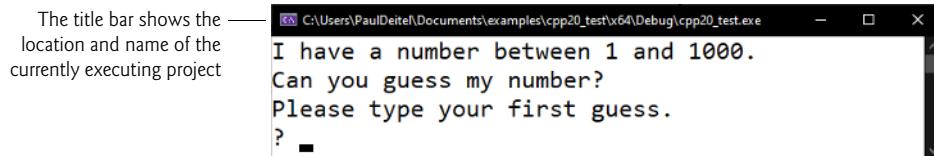
Step 5: Configuring Your Project to Use C++20

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

1. Right-click the project's node—`cpp20_test`—in the **Solution Explorer** and select **Properties** to display the project's `cpp20_test` **Property Pages** dialog.
2. In the **Configuration** drop-down list, select **All Configurations**. In the **Platform** drop-down list, select **All Platforms**.
3. In the left column below **Configuration Properties**, expand the **C/C++** node, then select **Language**.
4. In the right column, click in the field to the right of **C++ Language Standard**, click the down arrow, then select **ISO C++20 Standard (/std:c++20)**⁶⁵ and click **OK**.

Step 6: Compiling and Running the Project

To compile and run the project so you can test-drive the guess-the-number game, 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:

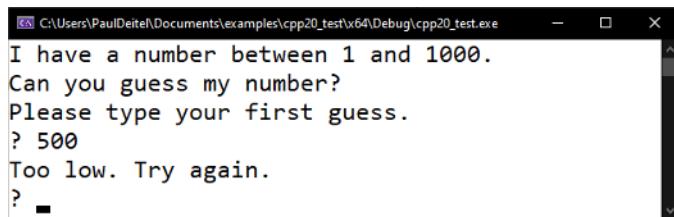


Step 7: Entering Your First Guess

At the ? prompt, type **500**, and press *Enter*—the program selects a number randomly each time you run it, so your interactions with the program will vary during each execution. In our case, the application displayed "Too low. Try again." to indicate the value was less than the number the application chose as the correct guess:

64. For the multiple source-code-file programs that you'll see in later chapters, select all the files for a given program.

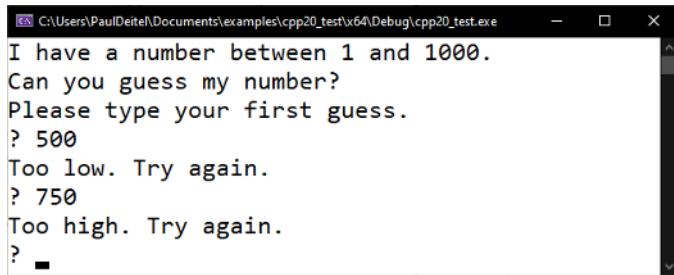
65. To try new C++ features as Microsoft adds them to Visual C++, select **Preview - Features from the Latest C++ Working Draft (/std:c++latest)** rather than **ISO C++20 Standard (/std:c++20)**.



```
C:\Users\PaulDeitel\Documents\examples\cpp20_test\x64\Debug\cpp20_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
?
```

Step 8: Entering Another Guess

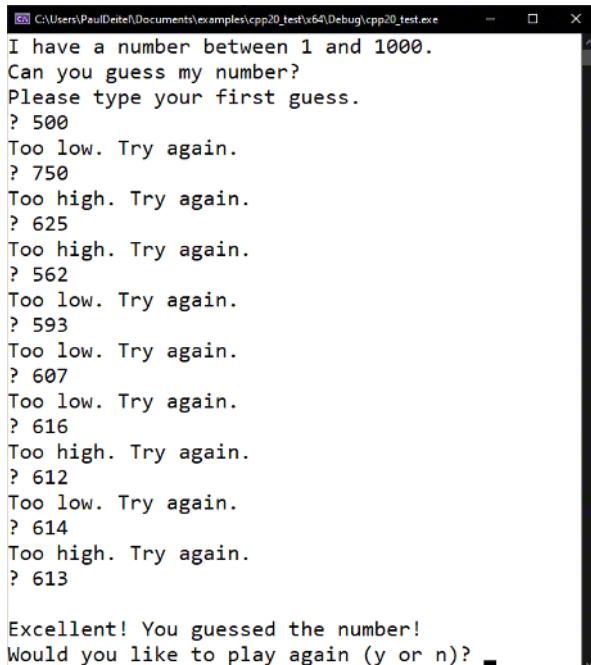
At the next prompt, if your system said the first guess was too low, type **750** and press *Enter*; otherwise, type **250** and press *Enter*. In our case, we entered **750**, and the application displayed "Too high. Try again." because the value was greater than the correct guess:



```
C:\Users\PaulDeitel\Documents\examples\cpp20_test\x64\Debug\cpp20_test.exe
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 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!"



```
C:\Users\PaulDeitel\Documents\examples\cpp20_test\x64\Debug\cpp20_test.exe
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 high. Try again.
? 625
Too high. Try again.
? 562
Too low. Try again.
? 593
Too low. Try again.
? 607
Too low. Try again.
? 616
Too high. Try again.
? 612
Too low. Try again.
? 614
Too high. Try again.
? 613

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 guessing the correct number (which you should always be able to do in 10 or fewer guesses), 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.

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 program. 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. 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.

Using Ubuntu Linux in the Windows Subsystem for Linux

Some Windows users may want to use the GNU g++ compiler on Windows. You can do this using the **GNU Compiler Collection Docker container** (Section 1.11.3), or you can use g++ in Ubuntu Linux running in the **Windows Subsystem for Linux**. To install the Windows Subsystem for Linux, follow the instructions at

<https://docs.microsoft.com/en-us/windows/wsl/install>

Once you install and launch the **Ubuntu** app on your Windows System, you can use the following command to change to the folder containing the test-drive code example on your Windows system:

`cd /mnt/c/Users/YourUserName/Documents/examples/ch01`

Then you can continue with *Step 2* in Section 1.11.2.

1.11.2 Compiling and Running with GNU C++ on Linux

In this section, you'll run a C++ program in a Linux shell using the GNU C++ compiler (g++).⁶⁶ For this test-drive, we assume that you've read the **Before You Begin** section and that you've placed the book's examples in your user account's `Documents/examples` folder.

Step 1: Changing to the ch01 Folder

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

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

66. At the time of this writing, the current g++ version was 13.1. You can determine your system's g++ version number with the command `g++ --version`. If you have an older version of g++, consider searching online for the instructions to upgrade the GNU Compiler Collection (GCC) for your Linux distribution or consider using the GCC Docker container discussed in Section 1.11.3.

In this section's figures, we use **bold** to highlight the user inputs. 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 with the `g++` command:⁶⁷

- The `-std=c++20` option indicates that we're using C++20.
- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program. If you do not include this option, `g++` automatically names the executable `a.out`.

```
~/Documents/examples/ch01$ g++ -std=c++20 GuessNumber.cpp -o GuessNumber  
~/Documents/examples/ch01$
```

Step 3: Running the Application

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

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

The `./` before `GuessNumber` tells Linux to run `GuessNumber` from the current directory.

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**—the program selects a number randomly each time you run it, so your interactions with the program will vary during each execution:

```
~/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.  
?
```

In our case, the application displayed "Too high. Try again." because the value entered was greater than the number the application chose as the correct guess.

⁶⁷. If you have multiple `g++` versions installed, you might need to use `g++-##`, where ## is the `g++` version number. For example, the command `g++-13` might be required to run `g++` version 13.

Step 5: Entering Another Guess

At the next prompt, if your system said the first guess was too low, type **750** and press *Enter*; otherwise, type **250** and press *Enter*. In our case, we entered **250**, and the application displayed "Too high. Try again." because the value was greater than the correct 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.
? 250
Too high. Try again.
?
```

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.":

```
? 125
Too high. Try again.
? 62
Too low. Try again.
? 93
Too low. Try again.
? 109
Too high. Try again.
? 101
Too low. Try again.
? 105
Too high. Try again.
? 103
Too high. Try again.
? 102

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 guessing the correct number (which you should always be able to do in 10 or fewer guesses), 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.

1.11.3 Compiling and Running with g++ in the GCC Docker Container

You can use the latest GNU C++ compiler on your system, regardless of your operating 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 that you've already performed the following steps described in the **Before You Begin** section that follows the **Preface**:

1. Installed either Docker Desktop (Windows or macOS) or Docker Engine (Linux).
2. Installed the **GCC Docker container**.
3. Placed the book's examples in your user account's `Documents/examples` folder.

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 `cd` to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the GCC Docker container with the command⁶⁸
`docker run --rm -it -v "%CD%":/usr/src gcc:latest`
3. macOS/Linux users: Launch the GCC Docker container with the command
`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 compile and run programs using the GNU C++ compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) maps the Docker container's `/usr/src` folder to the folder from which you executed the `docker run` command on your local computer, enabling the Docker container to access files in your local folder. You'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples.
- `gcc:latest` is the container name. The `:latest` specifies that you want to use the most up-to-date version of the `gcc` container.⁶⁹

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

```
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 container's `/usr/src` folder. In the Docker container, use the `cd` command to change to the `ch01` subfolder of `/usr/src`:

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

To compile, run and interact with the `GuessNumber` application in the Docker container, follow *Steps 2–7* of Section 1.11.2, Compiling and Running with GNU C++ on Linux.

68. A notification might appear asking you to allow Docker to access the files in the current folder. You must allow this; otherwise, you will not be able to access our source-code files in Docker.

69. If you'd like to keep your GCC container up-to-date with the latest release, you can execute the command `docker pull gcc:latest` before running the container.

Terminating the Docker Container

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

1.11.4 Compiling and Running with clang++ in a Docker Container

As with g++, you can use the latest LLVM/Clang C++⁷⁰ (clang++) command-line compiler on your system, regardless of your operating system. Currently, the LLVM/Clang team does not have an official Docker container, but many working containers are available on <https://hub.docker.com>. This section assumes that you've already performed the following steps described in the **Before You Begin** section that follows the **Preface**:

1. Installed either **Docker Desktop** (Windows or macOS) or **Docker Engine** (Linux).
2. Installed the **GCC Docker container**.
3. Placed the book's examples in your user account's `Documents/examples` folder.

Executing the teeks99/clang-ubuntu Docker Container

Open a **Command Prompt** (Windows), **Terminal** (macOS/Linux) or **shell** (Linux), then perform the following steps to launch the **teeks99/clang-ubuntu** Docker container:

1. Use `cd` to navigate into the `examples` folder containing this book's examples.
2. Windows users: Launch the Docker container with the command⁷¹
`docker run --rm -it -v "%CD%":/usr/src teeks99/clang-ubuntu:16`
3. macOS/Linux users: Launch the Docker container with the command
`docker run --rm -it -v "$(pwd)":/usr/src teeks99/clang-ubuntu:16`

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 `clang++` compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) maps the Docker container's `/usr/src` folder to the folder from which you executed the `docker run` command on your local computer, enabling the Docker container to access files in your local folder. You'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples.
- `teeks99/clang-ubuntu:16` is the container name.

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

```
root@9753bace2e87:/#
```

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

70. The Clang C++ (clang++) Docker container is a good alternative to Xcode for macOS users. The version of Clang C++ used in Xcode is missing several key C++20 features used throughout this book.

71. A notification will appear asking you to allow Docker to access the files in the current folder. You must allow this; otherwise, you will not be able to access our source-code files in Docker.

Changing to the ch01 Folder in the Docker Container

The `docker run` command specified above attaches your `examples` folder to the container's `/usr/src` folder. In the Docker container, use the `cd` command to change to the `ch01` subfolder of `/usr/src`:

```
root@9753bace2e87:/# cd /usr/src/ch01
root@9753bace2e87:/usr/src/ch01#
```

Compiling the Application

Before running the application, you must first compile it. This container uses the command `clang++`, as in

```
clang++ -std=c++20 GuessNumber.cpp -o GuessNumber
```

where:

- The `-std=c++20` option indicates that we're using C++20.
- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program. If you do not include this option, `clang++` automatically names the executable `a.out`.

Running the Application

To run and interact with the `GuessNumber` application in the Docker container, follow *Steps 3–7* of Section 1.11.2, Compiling and Running with GNU C++ on Linux.

Terminating the Docker Container

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

1.11.5 Compiling and Running with Xcode on macOS

In this section, you'll run a C++ program on macOS using Apple's Xcode IDE.⁷² Xcode uses a version of the LLVM/Clang C++ compiler, which at the time of this writing, is missing several important C++20 features that we use throughout the book. Visit

<https://deitel.com/cpphttp11>

for instructions on using Xcode to run the book's examples in Chapters 4 and higher.

Step 1: Checking Your Setup

If you have not already done so, read the **Before You Begin** section to install Xcode and download the book's code examples.

Step 2: Launching Xcode

In the Finder's **Go** menu, select **Applications** then double-click the Xcode icon:



If this is your first time running Xcode, the **Welcome to Xcode** window appears. Close this window—you can access it by selecting **Window > Welcome to Xcode**. We use the **>** char-

72. Xcode version was 14.2 at the time of this writing.

acter to indicate selecting a menu item from a menu. For example, **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 (left), **Editor** area (middle) and **Utilities** area (right).

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 two of these navigators:

- **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. When you select a file in the **Project** navigator, the file's contents display in the **Editor** area. The right-side **Utilities** area typically displays **inspectors**. For example, if you were building an iPhone app that contained a touchable button, you'd be able to configure the button's properties (its label, size, position, etc.) in this area. You will not use the **Utilities** area in this book. There's also a **Debug** area where you'll interact with the running guess-the-number program. This will appear below the **Editor** area.

The workspace window's toolbar contains options for executing a program (▶), and hiding or showing the left (Navigator, 📁) and right (Utilities, 📁) areas. It also contains a display area that shows the progress of tasks executing in Xcode.

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

The Apple 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 **Editor** area's left side, select your project's name under **TARGETS**.
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 **GNU++20** [**-std=gnu++20**].

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

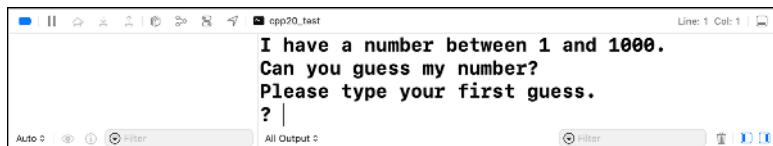
By default, Xcode creates a `main.cpp` source-code file containing a simple program that displays "Hello, World!". You won't use `main.cpp` in this test-drive, so you should delete the file. In the **Project** navigator, right-click the `main.cpp` file and select **Delete**. In the dialog that appears, 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 `cpp20_test` folder in the **Project** navigator. In the dialog that appears, ensure that **Copy items if needed** is checked, then click **Finish**.⁷³

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, and the application displays "Please type your first guess." and a question mark (?) as a prompt for input:



Step 8: Entering Your First Guess

Click the **Debug** area, then type `500` and press *Return*—the program selects a number randomly each time you run it, so your interactions with the program will vary during each execution. In our case, the application displayed "Too high. Try again." because the value was more than the number the application chose as the correct guess.

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

Step 9: Entering Another Guess

At the next prompt, if your system said the first guess was too low, type `750` and press *Enter*; otherwise, type `250` and press *Enter*. In our case, we entered `250`, and the application displayed "Too high. Try again." because the value was greater than the correct guess:

73. 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.

```
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 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.":

```
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 low. Try again.  
? 187  
Too high. Try again.  
? 156  
Too high. Try again.  
? 140  
Too low. Try again.  
? 148  
Too high. Try again.  
? 144  
Too high. Try again.  
? 142  
Too low. Try again.  
? 143  
  
Excellent! You guessed the number!  
Would you like to play again (y or n)? |
```

Playing the Game Again or Exiting the Application

After guessing the correct number (which you should always be able to do in 10 or fewer guesses), 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.

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 **Remove Reference**. You can then repeat *Step 6* to add a different program to the project. When you begin creating programs yourself, you can add a new empty file to the project as follows:

1. Right-click the folder in your project and select **New File....**
2. Select the **Empty** file template from the **Other** category and click **Next**.
3. Specify a name for your file and be sure to include the **.cpp** extension, then click **Create**.

1.12 Internet, World Wide Web, the Cloud and IoT

In the late 1960s, ARPA—the Advanced Research Projects Agency of the United States Department of Defense—rolled out plans for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at speeds on the order of 50,000 bits per second, a stunning rate at a time when most people (of the few who even had networking access) were connecting over telephone lines to computers at a rate of 110 bits per second. Academic research was about to take a giant leap forward. ARPA implemented what quickly became known as the ARPANET, the precursor to today's **Internet**. Today's fastest Internet speeds are on the order of billions of bits per second. Trillion-bits-per-second (terabit) speeds are already in service—the Energy Sciences Network in the United States is now capable of transmitting data at 46 Terabits per second!⁷⁴

Things worked out differently from the original plan. Although the ARPANET enabled researchers to network their computers, its main benefit proved to be the capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today's Internet, with e-mail, instant messaging, file transfer and social media, such as Snapchat, TikTok, Instagram, Facebook and Twitter, enabling billions of people worldwide to communicate quickly and easily.

The protocol (set of rules) for communicating over the ARPANET became known as the **Transmission Control Protocol (TCP)**. TCP ensured that messages, consisting of sequentially numbered pieces called **packets**, were properly delivered from sender to receiver, arrived intact and were reassembled in the correct order.

1.12.1 The Internet: A Network of Networks

In parallel with the evolution of the ARPANET, organizations worldwide were implementing their own networks for intra-organization (that is, within an organization) and inter-organization (that is, between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these different networks to communicate with each other. ARPA accomplished this by developing the **Internet Protocol (IP)**, which created a true “network of networks,” the Internet’s current architecture. The combined set of protocols is now called **TCP/IP**. Each Internet-connected device has an **IP address**—a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

Businesses rapidly realized that, by using the Internet, they could improve their operations and offer new and better services to their clients. Companies started spending large amounts of money on developing and enhancing their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. As a result, **Internet bandwidth**—the information-carrying capacity of communications lines—has increased tremendously while hardware costs have plummeted.

74. Michael Irving, “World's fastest Internet network upgraded to staggering 46 Terabit/s.” October 11, 2022. Accessed March 18, 2023. <https://newatlas.com/telecommunications/esnet6-worlds-fastest-internet-46-terabit-second/>.

1.12.2 The World Wide Web: Making the Internet User-Friendly

The **World Wide Web** (simply called “the web”) is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began developing **HyperText Markup Language (HTML)**—the technology for sharing information via “hyperlinked” text documents. He also wrote communication protocols such as **HyperText Transfer Protocol (HTTP)** to form the backbone of his new hypertext information system, which he referred to as the World Wide Web.

In 1994, Berners-Lee founded the **World Wide Web Consortium (W3C, <https://www.w3.org>)**, which was devoted to developing web technologies. A key W3C goal is to make the web universally accessible to everyone regardless of disabilities, language or culture.

1.12.3 The Cloud

More and more computing today is done “in the cloud”—that is, using software and data distributed across the Internet worldwide rather than locally on your desktop, notebook computer or mobile device. **Cloud computing** allows you to increase or decrease computing resources to meet your needs at any given time, which is more cost-effective than purchasing your own hardware to provide enough storage and processing power to meet occasional peak demands. Cloud computing also saves money by shifting to the service provider the burden of managing computing resources (such as installing and upgrading the software, security, backups and disaster recovery).

The apps you use daily are heavily dependent on various **cloud-based services**. These use massive clusters of computing resources (computers, processors, memory, storage, etc.) and databases that communicate over the Internet with each other and the apps you use. A service that provides access to itself over the Internet is known as a **web service**. As of 2023, Amazon Web Services (AWS; 33% market share), Microsoft Azure (22%) and Google Cloud Platform (GCP; 11%) account for about two-thirds of the cloud-services market.⁷⁵ Other top cloud vendors include IBM Cloud, Oracle, VMWare, Salesforce and Alibaba.⁷⁶

1.12.4 The Internet of Things (IoT)

The Internet is no longer just a network of computers—it’s an **Internet of Things (IoT)**. A thing is any device with an IP address and the ability to send, and in some cases receive, data automatically over the Internet. Such things include:

- cars with transponders for paying tolls,
- monitors for parking-space availability in a garage,
- heart monitors implanted in humans,
- water-quality monitors,
- smart meters that report energy usage,
- radiation detectors,

75. Charles Griffiths, “The Latest Cloud Computing Statistics.” March 6, 2023. Accessed March 25, 2023. <https://aag-it.com/the-latest-cloud-computing-statistics/>.

76. Anina Ot, “Top 16 Cloud Service Providers & Companies in 2023.” February 22, 2023. Accessed March 25, 2023. <https://www.datamation.com/cloud/cloud-service-providers/>.

- item trackers in warehouses,
- mobile apps that can track your movements,
- smart thermostats that adjust room temperatures based on weather forecasts and activity in the home,
- intelligent home appliances, and
- many more.

According to IOT Analytics, which provides market research and insights for the Internet of Things, there were 12.2 billion active IoT devices in 2021, and that is expected to grow to 27 billion in 2025.⁷⁷

1.12.5 Edge Computing

IoT is an example of **edge computing**,⁷⁸ in which data processing and storage occurs on or close to the network-connected devices that produce the data—that is, at the edge of the network. Edge computing often is used in “intelligent applications that require continuous monitoring and rapid response to dynamic conditions or high-volume data streams.”⁷⁹ It enables devices to process data and make decisions locally. For example, if a self-driving car loses its Internet connection, it must still be able to make the decisions that enable it to drive safely. Edge computing is used in many fields, including autonomous vehicles, smart power grids, healthcare devices, agriculture, clean energy, industrial automation and more.

1.12.6 Mashups

The applications-development methodology of **mashups** enables you to rapidly develop powerful software by combining (often free) complementary web services and various forms of information feeds. One of the first mashups, www.housingmaps.com, combined the real-estate listings from www.craigslist.org with Google Maps to show the locations of homes for sale or rent in a given area. Check out www.housingmaps.com for some interesting facts, history and articles on how it influenced real-estate industry listings.

A focus of this book’s object-natural approach is software reuse—you’ll work with easy-to-use, powerful preexisting classes that do significant things. Similarly, before building apps that use web services, consider reusing the many powerful preexisting services available across the Internet. Numerous online catalogs exist listing dozens to thousands of existing web services, including:

- GitHub’s Public APIs page (<https://github.com/public-apis/public-apis>) lists 51 categories containing hundreds of publicly available web services.
- The Google APIs Explorer (<https://developers.google.com/apis-explorer>) enables you to learn about and experiment with 266 of Google’s web services without writing any code.

77. Mohammad Hasan, “State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally.” May 18, 2022. Accessed March 18, 2023. <https://iot-analytics.com/number-connected-iot-devices/>.

78. “Edge computing.” Wikipedia. Wikimedia Foundation. Accessed March 25, 2023. https://en.wikipedia.org/wiki/Edge_computing.

79. Robert Napoli, “Edge Computing: What Is It And Why Does It Matter?” April 25, 2022. Accessed March 25, 2023. <https://www.forbes.com/sites/forbestechcouncil/2022/04/25/edge-computing-what-is-it-and-why-does-it-matter/>.

- Rapid (<https://rapidapi.com>) claims to be the world's largest API hub, listing thousands of free and paid APIs. Rapid also provides tools to help companies build and distribute their own APIs.
- APILayer (<https://apilayer.com>) lists 94 web services. All have free tiers with no credit card required.
- Any API (<https://any-api.com/>) lists over 1400 web services in 34 categories.



Checkpoint

1 *(Fill-In)* The _____ (simply called “the web”) is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos).

Answer: World Wide Web.

2 *(Fill-In)* In the Internet of Things (IoT), a thing is any object with a(n) _____ and the ability to send, and in some cases receive, data over the Internet.

Answer: IP address.

1.13 Metaverse

The term **Metaverse** was coined in Neal Stephenson’s 1992 novel *Snow Crash*, where the character Hiro “exists in a computer-generated universe that his computer is drawing into his goggles and pumping into his earphones. ...this imaginary place is known as the Metaverse.”^{80,81} Gartner says the Metaverse is the Internet’s next iteration and defines it as a “collective virtual shared space, created by the convergence of virtually enhanced physical and digital reality.”⁸² This section presents several key Metaverse technologies. Section 1.16 presents artificial intelligence, which improves the Metaverse’s immersive nature by helping generate realistic digital avatars, environments, natural-language dialog in your preferred spoken language and more.⁸³

1.13.1 Virtual Reality (VR)

Virtual reality (VR) typically refers to simulated, three-dimensional (3D) environments that you interact with using:

- non-immersive VR—in which you view the 3D environment in two-dimensional space using your computer’s screen,
- semi-immersive VR—in which you view the 3D environment in two-dimensional space using large displays, such as big-screen TVs or video walls, or

80. Neal Stephenson. *Snow Crash*. Bantam Books, 1993. p. 24.

81. “Metaverse.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Metaverse>.

82. Jackie Wiles, “What Is a Metaverse? And Should You Be Buying In?” October 21, 2022. Accessed March 18, 2023. <https://www.gartner.com/en/articles/what-is-a-metaverse>.

83. Hanna Hryshkevich, “How AI and the Metaverse Work Together.” February 2, 2023. Accessed March 27, 2023. <https://www.aitimejournal.com/how-ai-and-the-metaverse-work-together/>.

- fully immersive VR—in which you interact with the 3D environment via a head-mounted display (like a headset) and other devices (like hand-held controllers), making you feel like you're part of the environment.

1.13.2 Augmented Reality (AR)

Augmented reality (AR) projects real-time computer-generated content—such as images, sound, video, animation and more—onto what you see in the real world, typically via devices with built-in cameras, like smartphones, tablets and computerized glasses that enable you to see both the real world and virtual content at the same time.^{84,85}

One of the most popular examples of augmented reality is the mobile game app Pokémon Go™ in which Pokémon characters are projected onto real-world locations that you find via the GPS in your mobile devices.⁸⁶

Other uses include:

- In sports like baseball and hockey, broadcasters help you follow the baseball or puck by displaying a line that traces the path of a baseball pitch or the path of a hockey puck as it's passed around the ice.
- National Football League® broadcasts display colored lines over the field, showing the line of scrimmage and where the team needs to reach for a first down.
- Navigation apps can display real-time navigation and traffic information onto real-world maps and street-level views.
- Home decoration apps can show you what a room would look like after you repaint it or add specific furniture items.
- Clothing retailers often can show you what you'd look like in a particular clothing item you're considering purchasing.
- Museums can make exhibits more interactive by displaying information about a specific exhibit you are viewing.

Many companies are working on AR glasses and headsets.⁸⁷ For example, at the 2022 Google I/O conference, Google unveiled their AR glasses that translate text in real-time and project the translations onto the lenses.⁸⁸

-
84. Alexander S. Gillis, "What is augmented reality (AR)??" Accessed March 18, 2023. <https://www.techtarget.com/whatis/definition/augmented-reality-AR>.
 85. "Augmented reality." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Augmented_reality.
 86. Nick Statt, "Niantic adds 'reality blending' to Pokémon Go to make your virtual pals even more realistic." May 26, 2020. Accessed March 18, 2023. <https://www.theverge.com/2020/5/26/21269862/niantic-pokemon-go-reality-blending-ar-features-release-update>.
 87. Michael Sawh. "Best smartglasses and AR specs 2023: Tested picks from Snap, Meta and Amazon." November 15, 2022. Accessed March 18, 2023. <https://www.wearable.com/ar/the-best-smartglasses-google-glass-and-the-rest>.
 88. Mike Elgan, "What's so great about Google's 'translation glasses?'" May 20, 2022. Accessed March 18, 2023. <https://www.computerworld.com/article/3661209/what-s-so-great-about-google-s-translation-glasses.html>.

1.13.3 Mixed Reality

Mixed reality is similar to augmented reality but takes it further by enabling you to interact with the virtual objects projected into your real-world view. In that sense, mixed reality combines aspects of both augmented reality and virtual reality. Like virtual reality, the best mixed-reality experiences typically require a headset.^{89,90} The term **extended reality** encompasses virtual reality, augmented reality and mixed reality.⁹¹

1.13.4 Blockchain

Accounting systems use ledgers to keep track of transactions. **Blockchain**^{92,93} is a software-based ledger distributed over nodes on the Internet. It maintains transaction data in cryptographically secured chunks—called blocks—that are linked together, and new blocks are always added at the end. Once created, each block is immutable (not modifiable), so the blockchain maintains a permanent transaction record. The blockchain is decentralized, so no one person or group has control. This decentralization is the basis for the **decentralized apps (dApps)** that enable digital ownership in the Metaverse.

Blockchain is behind various digital technologies that have exploded in popularity over the last several years, such as cryptocurrencies (Section 1.13.5) and non-fungible tokens (Section 1.13.7). It's also used to implement many other kinds of applications,⁹⁴ including

- decentralized finance (DeFi) applications (lending, money transfer),
- smart contracts,
- safer Internet of Things (IoT) apps,
- secure digital identities to help prevent fraud,
- securing and sharing healthcare information,
- business logistics, and
- protecting intellectual property (IP).

1.13.5 Bitcoin and Cryptocurrency

“A **cryptocurrency** is a digital or virtual currency secured by cryptography.”⁹⁵ Cryptocurrency (or Crypto) is stored in digital wallets and generally uses blockchain technology for

-
89. Bernard Marr, “The Important Difference Between Augmented Reality And Mixed Reality.” Accessed March 18, 2023. <https://bernardmarr.com/the-important-difference-between-augmented-reality-and-mixed-reality/>.
90. “Mixed reality.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Mixed_reality.
91. “Extended reality.” Wikipedia. Wikimedia Foundation. Accessed March 25, 2023. https://en.wikipedia.org/wiki/Extended_reality.
92. “Blockchain.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Blockchain>.
93. Xiaojie Liu, “What Is a Blockchain?” Accessed March 18, 2023. <https://www.investopedia.com/terms/b/blockchain.asp>.
94. Sam Daley, “33 Blockchain Applications and Real-World Use Cases.” March 2, 2023. Accessed March 18, 2023. <https://builtin.com/blockchain/blockchain-applications>.
95. Jake Frankenfield, “Cryptocurrency Explained With Pros and Cons for Investment.” February 4, 2023. Accessed March 18, 2023. <https://www.investopedia.com/terms/c/cryptocurrency.asp>.

secure, decentralized transaction tracking and processing.⁹⁶ The original, most well-known and most valuable cryptocurrency is Bitcoin, which was created in 2009.⁹⁷ It is a decentralized cryptocurrency built using blockchain technology.⁹⁸

1.13.6 Ethereum

Ethereum is a programmable platform that enables you to “build and deploy decentralized applications (dApps) on its network … that use the blockchain to store data or control what your app can do.” One of its key features is **smart contracts**, which are used to implement lending apps, decentralized trading exchanges, insurance, crowdfunding apps and more.⁹⁹ Its native cryptocurrency is Ether—the world’s second most valuable cryptocurrency.¹⁰⁰

1.13.7 Non-Fungible Tokens (NFTs)

Non-fungible tokens (NFTs)¹⁰¹ are blockchain-based “cryptographically unique tokens that are linked to digital or physical content, providing proof of ownership.” They’re commonly used to enable individuals to own unique items, such as digital or physical artwork and photographs; limited edition digital collectibles; virtual real estate and goods in games and the Metaverse; physical real estate; intellectual property; and digital books.¹⁰²

Some artwork NFTs have sold for massive amounts of money:¹⁰³

- The most expensive NFT ever sold was Pak’s “The Merge,” purchased by 29,983 individual buyers for \$91.8 million—each buyer owns a small portion of the NFT.
- The most expensive NFT ever sold to one buyer was “Everydays: The First 5000 Days” by digital artist Mike “Beeple” Winkelmann, which sold for \$69.3 million.

1.13.8 Web3

Like the Metaverse, some people refer to **Web3** as the Internet’s next generation.^{104,105} Web3 and the Metaverse are sometimes confused with one another because they share some of the same technologies. Forbes defines Web3 as “the decentralized Internet—built on distributed technologies like blockchain … rather than centralized on servers owned by

-
96. “Cryptocurrency.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Cryptocurrency>.
 97. Julie Pinkerton, “The History of Bitcoin, the First Cryptocurrency.” February 27, 2023. Accessed March 18, 2023. <https://money.usnews.com/investing/articles/the-history-of-bitcoin>.
 98. “Bitcoin.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Bitcoin>.
 99. “What is Ethereum?” Accessed March 18, 2023. <https://ethereum.org/en/what-is-ethereum/>.
 100. “Ethereum.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Ethereum>.
 101. “Non-fungible token.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Non-fungible_token.
 102. Melanie Kramer, Stephen Graves and Daniel Phillips, “Beginner’s Guide to NFTs: What Are Non-Fungible Tokens?” Jan 12, 2022. Accessed March 18, 2023. <https://decrypt.co/resources/non-fungible-tokens-nfts-explained-guide-learn-blockchain>.
 103. “The 26 most expensive sold NFTs in the world.” Accessed March 25, 2023. <https://metav.rs/blog/most-expensive-nfts/>.
 104. Thomas Stackpole, “What Is Web3?” Harvard Business Review, May 10, 2022. Accessed March 18, 2023. <https://hbr.org/2022/05/what-is-web3>.
 105. “Web3.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Web3>.

individuals or corporations.”¹⁰⁶ A key focus of Web3 is that blockchain and its related technologies (like NFTs, cryptocurrencies and smart contracts) will give users greater control of their data and online interactions.



Checkpoint

1 (Fill-In) The _____, with its immersive user experiences, is considered by many to be the next iteration of the Internet.

Answer: Metaverse.

2 (Fill-In) _____ is the technology that serves as the basis for the decentralized apps (dApps) that enable digital ownership in the Metaverse.

Answer: Blockchain.

I.14 Software Development Technologies

As you develop software, you’ll frequently encounter the following buzzwords:

- **Refactoring:** Reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality and sometimes improving performance. Many IDEs contain built-in refactoring tools to do major portions of the reworking automatically.¹⁰⁷
- **Design patterns:** Proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to reuse them to develop better-quality software faster, with less money, time and effort.¹⁰⁸
- **Software Development Kits (SDKs):** The tools, libraries and documentation developers use to program applications. For example, SDKs can help you create applications for specific operating systems, gaming platforms, cloud platforms, and more.¹⁰⁹
- **Version Control Systems (VCS):** Tools that enable you to manage and keep track of the changes you make in software as you develop it over time. According to StackOverflow’s 2022 Developer Survey, Git—the technology behind GitHub—is by far the most widely used version control system, with almost 94% of developers saying they use it.¹¹⁰
- **Agile software development:** A software-development process that focuses on developing high-quality software iteratively and incrementally, with constant

106. Bernard Marr, “The Important Difference Between Web3 And The Metaverse.” February 22, 2022. Accessed March 25, 2023. <https://www.forbes.com/sites/bernardmarr/2022/02/22/the-important-difference-between-web3-and-the-metaverse/>.

107. “Code refactoring.” Wikipedia. Wikimedia Foundation. Accessed March 27, 2023. https://en.wikipedia.org/wiki/Code_refactoring.

108. “Software design pattern.” Wikipedia. Wikimedia Foundation. Accessed March 27, 2023. https://en.wikipedia.org/wiki/Software_design_pattern.

109. “Software development kit.” Wikipedia. Wikimedia Foundation. Accessed March 27, 2023. https://en.wikipedia.org/wiki/Software_development_kit.

110. “2022 Developer Survey.” May 2022. Accessed March 25, 2023. <https://survey.stackoverflow.co/2022/#version-control-version-control-system>.

interaction between the development team and customers. The process is geared toward enabling customers to begin using the software faster, even if it does not yet meet all their requirements or those requirements change.¹¹¹

- **Test-Driven Development (TDD):** An aspect of agile software development in which developers create software test-cases before creating the software itself. Then, as they develop the software, they continually ensure it passes all the test cases, enabling them to confirm the software meets the system requirements.¹¹²
- **Microservices:** A software architecture consisting of small, independent software services that can be combined to form larger applications. Microservices are often implemented as web services.¹¹³
- **Containerization:** In Sections 1.11.3 and 1.11.4, we discussed running the `g++` and `clang++` compilers in Docker containers. Containers are configured with everything you need to use their software, giving you a convenient mechanism to run that software across platforms. Containerization uses a technique called virtualization to hide the underlying platform details, enabling containers to execute across platforms.¹¹⁴ Docker is the most popular containerization platform in use today.¹¹⁵



Checkpoint

I *(Fill-In)* _____ is the process of reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality.

Answer: refactoring.

1.15 How Big Is Big Data?

For computer scientists and data scientists, understanding how to obtain, store, process and extract value from data is as crucial as writing effective programs. According to a 2016 IBM article, approximately 2.5 quintillion bytes (2.5 exabytes) of data are created daily,¹¹⁶ and 90% of the world's data was created in the previous two years.¹¹⁷ The Internet is responsible for much of this data generation. According to IDC, the global data supply will reach 175 zettabytes (equal to 175 trillion gigabytes or 175 billion terabytes) annually by 2025.¹¹⁸ Consider the following real-world examples of various popular data measures.

111. "Agile software development." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Agile_software_development.

112. "Test-driven development." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Test-driven_development.

113. "Microservices." Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/Microservices>.

114. "Containerization (computing)." Wikipedia. Wikimedia Foundation. Accessed March 25, 2023. [https://en.wikipedia.org/wiki/Containerization_\(computing\)](https://en.wikipedia.org/wiki/Containerization_(computing)).

115. "Containerization." Accessed March 25, 2023. <https://6sense.com/tech/containerization>.

116. "Welcome to the world of A.I.." Accessed March 18, 2023. <https://www.ibm.com/blogs/watson/2016/06/welcome-to-the-world-of-a-i/>.

117. "Accelerate Research and Discovery." Accessed March 18, 2023. <https://www.ibm.com/watson/advantages/accelerate>.

118. Andy Patrizio, "IDC: Expect 175 zettabytes of data worldwide by 2025." December 3, 2018. Accessed March 18, 2023. <https://www.networkworld.com/article/3325397/storage/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html>.

Megabytes

One megabyte (MB) is about one million (actually 2^{20}) bytes. High-quality MP3-format audio files require from 1 to 2.4 MB per minute.¹¹⁹ JPEG-format photos taken on a digital camera can require 8 to 10 MB per photo. On one of our iPhones, the **Camera** settings app reports that 1080p video at 30 frames-per-second (FPS) requires 65 MB/minute, and 4K video at 60 FPS requires 440 MB/minute.

Gigabytes

One gigabyte (GB) is 1024 megabytes. A DVD can store up to 8.5 GB¹²⁰, which could hold 141 hours of MP3 audio, 1000 photos from a 16-megapixel camera, 7.7 minutes of 1080p video at 30 frames-per-second (FPS), or 2.85 minutes of 4K video at 30 FPS. The current highest-capacity Ultra HD Blu-ray discs can store up to 100 GB of video.¹²¹ Streaming a 4K movie can use between 7 and 10 GB (highly compressed) per hour.

Terabytes

One terabyte (TB) is 1024 gigabytes. Recent disk drives for desktop computers come in sizes up to 20 TB,¹²² which could hold 28 years of MP3 audio, 1.68 million photos from a 16-megapixel camera, 226 hours of 1080p video at 30 frames-per-second (FPS), or 84 hours of 4K video at 30 FPS. Nimbus Data now has the largest solid-state drive (SSD) at 100 TB, which can store five times the 20 TB examples of audio, photos and video listed above.¹²³

Petabytes, Exabytes and Zettabytes

There are over four billion people online, creating about 2.5 quintillion bytes of data each day¹²⁴—that's 2500 petabytes (each petabyte is 1024 terabytes) or 2.5 exabytes (each exabyte is 1024 petabytes). A March 2016 *AnalyticsWeek* article stated that by 2021 there would be over 50 billion devices connected to the Internet (most of them through the Internet of Things; Section 1.12.4) and, by 2020, there would be 1.7 megabytes of new data produced per second for every person on the planet.¹²⁵ At today's population of 8.023 billion people¹²⁶, that's about 13.55 petabytes of new data per second, 813 petabytes per minute, 48,780 petabytes (48.78 exabytes) per hour, or 1171 exabytes per day—that is, 1.171 zettabytes (ZB) per day (each zettabyte is 1024 exabytes). That's the equivalent of 5.73 million hours of 4K video every day or approximately 121 billion photos every day!

119.“Audio File Size Calculations.” Accessed March 18, 2023. <https://www.audiomountain.com/tech/audio-file-size.html>.

120.“DVD.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/DVD>.

121.“Ultra HD Blu-ray.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Ultra_HD_Blu-ray.

122.“History of hard disk drives.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/History_of_hard_disk_drives.

123.Jakob Han, “Nimbus Data 100TB SSD – World’s Largest SSD.” March 29, 2018. Accessed March 18, 2023. <https://www.cined.com/nimbus-data-100tb-ssd-worlds-largest-ssd/>.

124.Aditya Rayaprolu, “How Much Data Is Created Every Day in 2020?” February 27, 2023. Accessed March 18, 2023. <https://techjury.net/blog/how-much-data-is-created-every-day/#gref>.

125.Vishal Kumar, “Big Data Facts.” Accessed March 18, 2023. <https://analyticsweek.com/content/big-data-facts/>.

126.“Current World Population.” Accessed March 18, 2023. <https://www.worldometers.info/world-population/>.

Additional Big-Data Stats

For a real-time sense of big data, check out <https://www.worldometers.info/>, with various statistics, including that day's numbers of Google searches, Tweets and e-mails sent. Some other interesting big-data facts:

- Every hour, YouTube users upload 30,000 hours of video, and almost 1 billion hours of video are watched on YouTube every day.¹²⁷
- Every second, there are 103,777 GBs (or 103.777 TBs) of Internet traffic, 9204 tweets sent, 87,015 Google searches and 86,617 YouTube videos viewed.¹²⁸
- On Facebook each day, there are 3.2 billion “likes” and comments,¹²⁹ and 5 billion emojis sent via Facebook Messenger.¹³⁰

Domo, Inc.’s infographic called “Data Never Sleeps” shows interesting statistics regarding how much data is generated every minute, including:¹³¹

- 66,000 photos posted on Instagram.
 - 500 hours of video uploaded to YouTube.
 - 1.7 million Facebook posts.
 - 2.43 million snaps on SnapChat.
 - 1 million hours of streaming video viewed.
 - 347,200 tweets sent on Twitter.
 - 104,600 hours spent in Zoom meetings.
- Processing the World’s Data Requires Lots of Electricity

A 2015 article stated that energy use for processing the world’s data was growing at 20% per year and consuming approximately 3–5% of the world’s power. The article said that total data-processing power consumption could reach 20% by 2025.¹³²

Another enormous electricity consumer is the blockchain-based cryptocurrency Bitcoin. Processing just one Bitcoin transaction uses approximately the same amount of energy as powering the average American home for a week. The energy use comes from the process Bitcoin “miners” use to prove that transaction data is valid.¹³³

Morgan Stanley predicted in 2018 that “the electricity consumption required to create cryptocurrencies this year could actually outpace the firm’s projected global electric

127. Kit Smith, “57 Fascinating and Incredible YouTube Statistics.” February 21, 2020. Accessed March 18, 2023. <https://www.brandwatch.com/blog/youtube-stats/>.

128. “Worldometer.” Accessed March 18, 2023. <https://www.worldometers.info/>.

129. Matt McGee. “Facebook: 3.2 Billion Likes & Comments Every Day.” Accessed March 18, 2023. <https://marketingland.com/facebook-3-2-billion-likes-comments-every-day-19978>.

130. Samantha Scelzo, “Facebook celebrates World Emoji Day by releasing some pretty impressive facts.” Accessed March 18, 2023. <https://mashable.com/2017/07/17/facebook-world-emoji-day/>.

131. “Data Never Sleeps.” Accessed March 18, 2023. <https://www.domo.com/data-never-sleeps>.

132. “‘Tsunami of data’ could consume one fifth of global electricity by 2025.” Accessed March 18, 2023. <https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025>.

133. “One Bitcoin Transaction Consumes As Much Energy As Your House Uses in a Week.” Accessed March 18, 2023. https://motherboard.vice.com/en_us/article/ywbbpm/bitcoin-mining-electricity-consumption-ethereum-energy-climate-change.

vehicle demand in 2025.”¹³⁴ This situation is unsustainable, especially given the huge interest in blockchain-based applications, even beyond the cryptocurrency explosion. The blockchain community is working on fixes.^{135,136}



Checkpoint

- 1 (Fill-In)** The technology that could wreak havoc with blockchain-based cryptocurrencies, like Bitcoin, and other blockchain-based technologies is _____.

Answer: quantum computers.

- 2 (True/False)** With cloud computing, you pay a fixed price for cloud services regardless of how much you use those services.

Answer: False. A key cloud-computing benefit is that you pay for only what you use to accomplish a given task.

1.15.1 Big-Data Analytics

The term “data analysis” was coined in 1962,¹³⁷ though people have been analyzing data using statistics for thousands of years, going back to the ancient Egyptians.¹³⁸ The term “big data” was coined around 1987.¹³⁹

Consider four of the “V’s of big data”:^{140,141}

1. Volume—the data the world is producing is growing exponentially.
2. Velocity—the speed at which data is being produced, the speed at which it moves through organizations and the speed at which data changes are growing quickly.^{142,143,144}

134. “Power Play: What Impact Will Cryptocurrencies Have on Global Utilities?” Accessed March 18, 2023. <https://www.morganstanley.com/ideas/cryptocurrencies-global-utilities>.

135. Mike Orcutt, “Blockchains Use Massive Amounts of Energy—But There’s a Plan to Fix That.” November 16, 2017. Accessed March 18, 2023. <https://www.technologyreview.com/s/609480/bitcoin-uses-massive-amounts-of-energybut-theres-a-plan-to-fix-it/>.

136. Stan Schroeder, “How to fix Bitcoin’s energy-consumption problem.” December 1, 2017. Accessed March 18, 2023. <http://mashable.com/2017/12/01/bitcoin-energy/>.

137. “A Very Short History Of Data Science.” Accessed March 18, 2023. <https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/>.

138. “A Brief History of Data Analysis.” Accessed March 18, 2023. <https://www.flydata.com/blog/a-brief-history-of-data-analysis/>.

139. Diebold, Francis. (2012). On the Origin(s) and Development of the Term “Big Data”. SSRN Electronic Journal. 10.2139/ssrn.2152421. https://www.researchgate.net/publication/255967292_On_the_Origins_and_Development_of_the_Term_.

140. “The Four V’s of Big Data – What is big data?” December 2021. Accessed March 18, 2023. <https://www.analyticsinsight.net/the-four-vs-of-big-data-what-is-big-data/>.

141. There are lots of articles and papers that add many other “V-words” to this list.

142. David Gewitz. “Volume, velocity, and variety: Understanding the three V’s of Big Data.” March, 21, 2018. Accessed March 18, 2023. <https://www.zdnet.com/article/volume-velocity-and-variety-understanding-the-three-vs-of-big-data/>.

143. Ben Lutkevich, “3Vs (volume, variety and velocity).” Accessed March 18, 2023. <https://what-is.techtarget.com/definition/3Vs>.

144. Brent Dykes, “Big Data: Forget Volume and Variety, Focus On Velocity.” June 28, 2017. Accessed March 18, 2023. <https://www.forbes.com/sites/brentdykes/2017/06/28/big-data-forget-volume-and-variety-focus-on-velocity>.

3. Variety—today's data includes characters, digits, punctuation and special characters, as well as images, audios, videos and data from an exploding number of Internet of Things sensors in our homes, businesses, vehicles, cities and more.
4. Veracity—the validity of the data—is it complete and accurate? Can we trust that data when making crucial decisions? Is it real?

Digital data storage has become so vast in capacity, and so cheap and small, that we can now conveniently and economically retain all the digital data we're creating.¹⁴⁵ That's big data. To get a sense of big data's scope in industry, government and academia, check out the high-resolution graphic (click it to zoom for easier readability)¹⁴⁶ at:

<https://mattturck.wpeenginepowered.com/wp-content/uploads/2021/12/2021-MAD-Landscape-v3.pdf>

1.15.2 Data Science and Big Data Are Making a Difference: Use Cases

The data-science field is growing rapidly because it produces results that make a difference in our personal and business lives. The following table enumerates lots of data-science and big-data use cases. Big-data analytics has resulted in improved profits, better customer relations, and even sports teams winning more games and championships.^{147,148,149}

Some data-science use cases		
assisting people with disabilities	human genome sequencing	ride-sharing
automated closed captioning	identity-theft prevention	robo financial advisors
brain mapping	improved inventory control	self-driving cars
cancer diagnosis/treatment	intelligent assistants	sentiment analysis
computer vision	location-based services	similarity detection
crime prevention	malware detection	smart homes
CRISPR gene editing	marketing analytics	smart meters
crop-yield improvement	medical device monitoring	smart thermostats
customer service agents	natural-language translation	smart traffic control
data visualization	personalized medicine	spam detection
diagnostic medicine	phishing elimination	stock market forecasting
dynamic driving routes	pollution reduction	summarizing text
electronic health records	precision medicine	theft prevention
emotion detection	preventative medicine	trend spotting
facial recognition	preventing disease outbreaks	visual product search
fraud detection	recommendation systems	voice recognition
health outcome improvement	reducing energy use	weather forecasting

145. Michael Lesk, "How Much Information Is There In the World?" Accessed March 18, 2023. <http://www.1esk.com/mlesk/ksg97/ksg.html>. [The following article pointed us to this Michael Lesk article: <https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/>.]

146. Matt Turck, "Red Hot: The 2021 Machine Learning, AI and Data (MAD) Landscape," <https://mattturck.com/data2021/>.

147. Sawchik, T., *Big Data Baseball: Math, Miracles, and the End of a 20-Year Losing Streak* (New York: Flat Iron Books, 2015).

148. Ayres, I., *Super Crunchers* (Bantam Books, 2007), pp. 7–10.

149. Lewis, M., *Moneyball: The Art of Winning an Unfair Game* (W. W. Norton & Company, 2004).

1.16 AI—at the Intersection of Computer Science and Data Science

When a baby first opens its eyes, does it “see” its parent’s faces? Does it understand any notion of what a face is—or even what a simple shape is? Babies must “learn” the world around them. That’s what artificial intelligence (AI) technology called machine learning does. It looks at massive amounts of data and learns from it. AI technologies are being used to play games, implement computer-vision applications, enable self-driving cars, enable robots to learn to perform new tasks, diagnose medical conditions, translate speech to other languages in near real-time, create chatbots that can respond smoothly and accurately to arbitrary questions after being trained on massive text databases called **large language models**, and much more. Who’d have guessed just a few years ago that artificially intelligent self-driving cars would be allowed on our roads—or even become common? The ultimate goal of the AI field is **artificial general intelligence**—the ability to perform any task that humans can.¹⁵⁰

1.16.1 Artificial Intelligence (AI)

Two definitions of **artificial intelligence (AI)**¹⁵¹ are:

- “a branch of computer science dealing with the simulation of intelligent behavior in computers” and
- “the capability of a machine to imitate intelligent human behavior.”

The term AI was coined by John McCarthy in 1956. AI-based systems now permeate our daily lives. Examples include computer vision, speech recognition, natural language translation, recommender systems (such as Amazon recommending products you might like or Netflix recommending shows and movies you might like), automated game-playing systems, self-driving cars, and more recent remarkable tools like the ChatGPT chatbot that carries on conversations almost indistinguishable from human conversations and the Dall-E 2 image generator that creates original images when given only text descriptions.¹⁵²

For many decades, AI has been a field with problems and no solutions. That’s because once a particular problem is solved, people say, “Well, that’s not intelligence; it’s just a computer program that tells the computer exactly what to do.” However, with AI techniques like machine learning (Section 1.16.4) and deep learning (Section 1.16.5), we’re not pre-programming solutions to specific problems. Instead, we’re having our computers solve problems by learning from data—and, typically, lots of it. Many of the most interesting and challenging problems are being pursued with deep learning. Google alone has more than a thousand deep-learning projects underway.^{153,154}

150. “Artificial general intelligence.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Artificial_general_intelligence.

151. “artificial intelligence.” Merriam-Webster.com. 2023. Accessed March 18, 2023. <https://www.merriam-webster.com/dictionary/artificial%20intelligence>.

152. “Artificial intelligence.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Artificial_intelligence.

153. Kelly Gonsalves, “Google has more than 1,000 artificial intelligence projects in the works.” Accessed March 18, 2023. <https://theweek.com/speedreads/654463/google-more-than-1000-artificial-intelligence-projects-works>.

154. Tiernan Ray, “Google says ‘exponential’ growth of AI is changing nature of compute.” Accessed March 18, 2023. <https://www.zdnet.com/article/google-says-exponential-growth-of-ai-is-changing-nature-of-compute/>.

1.16.2 Artificial General Intelligence (AGI)

Artificial general intelligence (AGI)¹⁵⁵ refers to artificial intelligence that can “carry out any task that humans are capable of.”^{156,157} Experts disagree on when or if artificial general intelligence might become a reality—from as soon as this decade, to sometime in the next 100 years, to never.^{158,159}

1.16.3 Artificial Intelligence Milestones

Several artificial intelligence milestones have captured people’s attention and imagination, made the general public start thinking that AI is real and made businesses think about commercializing it:

- In a 1997 match between **IBM’s DeepBlue** computer system and chess Grandmaster Gary Kasparov, DeepBlue became the first computer to beat a reigning world chess champion under tournament conditions.¹⁶⁰ IBM loaded DeepBlue with hundreds of thousands of grandmaster chess games. DeepBlue could use brute force to evaluate up to 200 million moves per second!¹⁶¹ This is big data at work. IBM received the Carnegie Mellon University Fredkin Prize, which in 1980 offered \$100,000 to the creators of the first computer to beat a world chess champion.¹⁶²
- In 2011, **IBM’s Watson** beat the two best human Jeopardy! players in a \$1 million match. Watson simultaneously used hundreds of language-analysis techniques to locate correct answers in 200 million pages of content (including all of Wikipedia), requiring four terabytes of disk storage.^{163,164} Watson was trained with AI machine-learning and reinforcement-learning (Section 1.16.6) techniques.¹⁶⁵

155. “Artificial general intelligence.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Artificial_general_intelligence.

156. Kerem Gülen, “How close are we to AI with human capabilities?” October 6, 2022. Accessed March 18, 2023. <https://dataconomy.com/2022/10/artificial-general-intelligence-definition/>.

157. “Technological singularity.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Technological_singularity.

158. Gary Marcus, “Artificial General Intelligence Is Not as Imminent as You Might Think.” July 1, 2022. Accessed March 25, 2023. <https://www.scientificamerican.com/article/artificial-general-intelligence-is-not-as-imminent-as-you-might-think1/>.

159. Max Roser, “AI timelines: What do experts in artificial intelligence expect for the future?” February 7, 2023. Accessed March 25, 2023. <https://ourworldindata.org/ai-timelines>.

160. “Deep Blue versus Garry Kasparov.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Deep_Blue_vs_Garry_Kasparov.

161. “Deep Blue (chess computer).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Deep_Blue_\(chess_computer\)](https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)).

162. “IBM Deep Blue Team Gets \$100,000 Prize.” Accessed March 18, 2023. <https://articles.latimes.com/1997/jul/30/news/mn-17696>.

163. Jo Best, “IBM Watson: The inside story of how the Jeopardy-winning supercomputer was born, and what it wants to do next.” Accessed March 18, 2023. <https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/>.

164. “Watson (computer).” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. [https://en.wikipedia.org/wiki/Watson_\(computer\)](https://en.wikipedia.org/wiki/Watson_(computer)).

165. Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., Schlaefler, N., & Welty, C. (2010). “Building Watson: An Overview of the DeepQA Project.” *AI Magazine*, 31(3), 59-79. <https://doi.org/10.1609/aimag.v31i3.2303>.

- Go—a board game created in China thousands of years ago¹⁶⁶—is widely considered one of the most complex games ever invented, with 10^{170} possible board configurations.¹⁶⁷ To understand how large that number is, it's believed there are (only) between 10^{78} and 10^{82} atoms in the known universe!¹⁶⁸ In 2015, **AlphaGo**—created by Google's DeepMind group—used the AI technique of deep learning (Section 1.16.5) with two neural networks to beat the European Go champion Fan Hui. Go is considered to be a far more complex game than chess.
- More recently, Google generalized its AlphaGo AI to create **AlphaZero**—a game-playing AI that teaches itself to play other games. In December 2017, AlphaZero learned the rules and taught itself to play chess in less than four hours using the AI technique of reinforcement learning. It then beat the world champion chess program, Stockfish 8, in a 100-game match—winning or drawing every game. After training itself in Go for just eight hours, AlphaZero was able to beat its AlphaGo predecessor in 60 of 100 games.¹⁶⁹

1.16.4 Machine Learning

Machine learning is “a subfield of artificial intelligence that gives computers the ability to learn without explicitly being programmed.”¹⁷⁰ Machine learning typically uses pre-defined algorithms to build models that learn from training data. Those models can then be used to make predictions.¹⁷¹ For example, given labeled training data that contains legitimate e-mails and junk (also called spam) e-mails, a machine-learning model can learn from the training data, then be used to classify e-mails it has never “seen” as junk or not junk. E-mail apps like Microsoft’s Outlook and Google’s Gmail use techniques like this to move junk emails out of your inbox.

1.16.5 Deep Learning

Deep learning is a subset of machine learning that involves building artificial neural networks, which use “algorithms inspired by the structure and function of the brain.”¹⁷² The term “deep” comes from the number of software “layers” in an artificial neural network—the more layers, the deeper the network. The network learns more from its training data as

-
166. “A Brief History of Go.” Accessed March 18, 2023. <http://www.usgo.org/brief-history-go>.
167. Nsikan Akpan, “Google artificial intelligence beats champion at world’s most complicated board game.” Accessed March 18, 2023. <https://www.pbs.org/newshour/science/google-artificial-intelligence-beats-champion-at-worlds-most-complicated-board-game>.
168. John Carl Villanueva, “How Many Atoms Are There in the Universe?” Accessed March 18, 2023. <https://www.universetoday.com/36302/atoms-in-the-universe/>.
169. Samuel Gibbs, “AlphaZero AI beats champion chess program after teaching itself in four hours.” Accessed March 18, 2023. <https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours>.
170. Sara Brown “Machine learning, explained.” April 21, 2021. Accessed March 18, 2023. <https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained>.
171. “Machine learning.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Machine_learning.
172. Jason Brownlee, “What is deep learning?” August 16, 2019. Accessed March 18, 2023. <https://machinelearningmastery.com/what-is-deep-learning/>.

it proceeds through the network layers.¹⁷³ The availability of big data, significant processor power, faster Internet speeds and advancements in parallel computing hardware and software are making it possible for more organizations and individuals to pursue resource-intensive deep-learning solutions that are producing impressive results in computer vision, speech recognition, speech synthesis and newer generative AI applications (Section 1.16.7).¹⁷⁴

1.16.6 Reinforcement Learning

Reinforcement learning¹⁷⁵ is a subset of machine learning in which algorithms learn from their environment, similar to how humans learn—for example, a video game enthusiast learning a new game or a baby learning to walk or recognize its parents. The algorithm implements an agent that learns by trying to perform a task, receiving feedback about success or failure, making adjustments then trying again. The goal is to maximize the reward. The agent receives a positive reward for doing a right thing and a negative reward (that is, a punishment) for doing a wrong thing. The agent uses this information to determine the next action to perform and must try to maximize the reward.

1.16.7 Generative AI—ChatGPT and Dall-E 2

Generative AI describes algorithms that can generate new content—including text, images, audio and video—based on prompts provided by a user as input.^{176,177}

ChatGPT

OpenAI's ChatGPT is an AI-based chatbot you can converse with using natural language.¹⁷⁸ For example, you could prompt it with text like

- “tell me about the history of C++,”
- “who coined the term Metaverse?” or
- “summarize the article at” followed by a link to the article.

ChatGPT then responds with answers that look and feel as if they were generated by a human. ChatGPT is based on a large language model named GPT (Generative Pre-Trained Transformer),¹⁷⁹ which was trained on massive amounts of text using several artificial intelligence techniques.¹⁸⁰ At the time of this writing, GPT-4 had just been released

173. “Deep learning.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Deep_learning.

174. Sharon Goldman, “10 years later, deep learning ‘revolution’ rages on, say AI pioneers Hinton, LeCun and Li.” September 14, 2022. Accessed March 27, 2023. <https://venturebeat.com/ai/10-years-on-ai-pioneers-hinton-lecun-li-say-deep-learning-revolution-will-continue/>.

175. M. Tim Jones, “Train a software agent to behave rationally with reinforcement learning.” October 11, 2017. Accessed March 27, 2023. <https://developer.ibm.com/articles/cc-reinforcement-learning-train-software-agent/>.

176. “What is generative AI?” January 19, 2023. Accessed March 18, 2023. <https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai>.

177. “Generative artificial intelligence.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Generative_artificial_intelligence.

178. “Introducing ChatGPT.” Accessed March 18, 2023. <https://openai.com/blog/chatgpt>.

179. “Large language model.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. https://en.wikipedia.org/wiki/Large_language_model.

180. “ChatGPT.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/ChatGPT>.

with the ability to process both text and image inputs.¹⁸¹ Powerful, large-language-model apps like ChatGPT and Google’s Bard¹⁸² sometimes produce incorrect results or “hallucinations,”¹⁸³ so you should verify the information tools like these provide.

When you converse with ChatGPT and similar tools, it may seem like you’re chatting with another person. In 1950, Alan Turing proposed the **imitation game**—now known as the **Turing test**—as a way to determine whether a computer exhibits intelligent behavior.¹⁸⁴ In it, three computer terminals would be hidden from one another, with two people operating terminals via text-based inputs and the third being computer-operated. One person would ask text-based questions that would be sent to the other two terminals. The second person and the computer would send text responses. If the person asking the questions could not determine whether the answers came from the other person or the computer, then the computer would be considered to have exhibited intelligent behavior.

Dall-E 2

OpenAI’s **Dall-E 2** is a generative AI for creating images from natural-language text prompts^{185,186} such as, “Generate an image of a golden retriever dog in the style of Claude Monet.” Visit <https://labs.openai.com/> to view sample images created by Dall-E 2—hovering over an image reveals the text prompt on which the image is based. Some text prompts that have produced cool-looking images include:

- “An armchair in the shape of an avocado.”
- “A futuristic neon lit cyborg face.”
- “A van Gogh style painting of an American football player.”
- “A Formula 1 car driving on a neon road.”
- “A Shiba Inu dog wearing a beret and black turtleneck.”
- “A comic book cover of a superhero wearing headphones.”



Checkpoint

1 *(Fill-In)* The ultimate goal of AI is to produce a(n) _____.

Answer: artificial general intelligence.

2 *(Fill-In)* IBM’s Watson beat the two best human Jeopardy! players. Watson was trained using a combination of _____ learning and _____ learning techniques.

Answer: machine, reinforcement.

181. “GPT-4” Accessed March 23, 2023. <https://openai.com/research/gpt-4>.

182. “Bard.” Accessed March 25, 2023. <https://bard.google.com/>.

183. Vilius Petkauskas, “ChatGPT’s answers could be nothing but a hallucination.” March 6, 2023. Accessed March 25, 2023. <https://cybernews.com/tech/chatgpts-bard-ai-answers-hallucination/>.

184. “Turing test.” Wikipedia. Wikimedia Foundation. Accessed March 25, 2023. https://en.wikipedia.org/wiki/Turing_test.

185. “Dall-E 2.” Accessed March 18, 2023. <https://openai.com/product/dall-e-2>.

186. “Dall-E.” Wikipedia. Wikimedia Foundation. Accessed March 18, 2023. <https://en.wikipedia.org/wiki/DALL-E>.

1.17 Wrap-Up

This chapter introduced basic computer hardware, software and Internet concepts. We considered how computers are organized into logical units. We discussed Moore's Law and noted that the continued improvements it so accurately predicted for many years seem to be ending. As a result, computer hardware designers have shifted to multi-core hardware architectures, which are programmed with the concurrent programming techniques we discuss in depth in Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, and Chapter 18, C++20 Coroutines.

We showed how data items can be viewed as an organized hierarchy of increasingly complex types, from bits, to bytes, to fields, to records to databases. We discussed the evolution of computer languages, from machine languages to assembly languages to high-level languages like C++, and we mentioned other popular high-level programming languages.

We considered popular desktop and laptop operating systems, including the proprietary Microsoft Windows and the open-source Linux. And we considered mobile operating systems, including Apple's proprietary iOS for its iPhones and Google's open-source Android that runs on a wide variety of smartphones from other vendors.

We encouraged you to avoid reinventing the wheel when developing new software and instead use existing software components from the C++ standard library and open-source C++ libraries, like Boost. We use Boost classes in two of our nine objects-natural case studies.

We presented a friendly introduction to object orientation using an automobile analogy to discuss classes, objects, inheritance and polymorphism. This provides the groundwork for our objects-natural case studies in Chapters 2 through 9 and our detailed treatment of the principles of object-oriented programming in Chapters 9–11, which are then used throughout the rest of the book.

We discussed a simplified view of a C++ development environment and the steps involved in C++ program development, including editing, preprocessing, compiling, linking, loading and executing. Between the software installation discussions in the Before You Begin section and the Test-Drives in Section 1.11, you installed the compiler(s) you'll use to compile this book's programs and the programs you write to solve the exercises.

We discussed the Internet, the World Wide Web, the Cloud and the Internet of Things (IoT). We introduced the Metaverse and associated technologies, including blockchain, cryptocurrencies, non-fungible tokens (NFTs), Web3, augmented reality (AR), virtual reality (VR) and mixed reality.

We considered various key software development technologies, including refactoring, design patterns, software development kits (SDKs), version control systems (VCS), agile software development, test-driven development (TDD), microservices and containerization.

We considered big data and various data sizes from megabytes to exabytes. Finally, we discussed artificial intelligence (AI)—an intersection between computer science and data science—introducing various AI technologies, including artificial general intelligence (AGI), machine learning and deep learning and reinforcement learning, and presenting various AI milestones, including generative AIs like ChatGPT and Dall-E 2. In Chapter 2, we begin our presentation of C++ programming.

Exercises

1.1 Categorize each of the following items as either hardware or software:

- a) CPU.
- b) C++ compiler.
- c) ALU.
- d) C++ preprocessor.
- e) input unit.
- f) an editor program.

1.2 (*Computer Organization*) Fill in the blanks in each of the following statements:

- a) The logical unit that receives information from outside the computer for use by the computer is the _____.
- b) _____ is the logical unit that sends information the computer has already processed to various devices for use outside the computer.
- c) _____ and _____ are the computer's logical units that retain information.
- d) _____ is the computer's logical unit for performing calculations.
- e) _____ is the computer's logical unit for making logical decisions.
- f) _____ is the computer's logical unit for coordinating the other logical units' activities.

1.3 Discuss the purpose of each of the following:

- a) `cin`
- b) `cout`
- c) `cerr`

1.4 Fill in the blanks in each of the following statements:

- a) Information in the computer's memory unit is _____—it's typically lost when the computer's power is turned off.
- b) As a measure of computer memory size, TB stands for _____—which is 1024 gigabytes.
- c) Most computers today have _____ processors that economically implement multiple processors on a single integrated circuit chip.
- d) _____ enable concurrent programming with a sequential like coding style.
- e) The most popular database model is the _____ database in which data is stored in simple tables of rows and columns.
- f) _____ is the process of searching through extensive collections of data to find valuable insights.
- g) With _____ software individuals contribute to developing, maintaining and evolving the software. Anyone can then use that software for their own purposes—normally at no charge, but subject to various (typically generous) licensing requirements.
- h) Today, most of the code for general-purpose operating systems is written in _____.
- i) The C++ programming models are _____, _____, _____, _____, _____.
- j) Existing older C++ code using older coding idioms is called _____ code.
- k) When writing new software avoid _____. Instead use existing pieces from code libraries.

- l) Microsoft's primary language for the Metaverse is _____.
- m) The _____ that objects come from are essentially reusable software components.
- n) The protocol for communicating over the ARPANET became known as the _____.
- o) TCP ensured that messages, consisting of sequentially numbered pieces called _____ were properly delivered from sender to receiver, arrived intact and were reassembled in the correct order.
- p) In the Internet of Things (IoT) a "thing" is any device with a(n) _____ and the ability to send, and sometimes receive, data automatically over the Internet.
- q) The applications-development methodology of _____ enables you to rapidly develop powerful software applications by combining (often free) complementary web services and various forms of information feeds.
- r) _____ typically refers to simulated, 3D environments that you interact with.
- s) _____ is a software-based ledger distributed over nodes on the Internet.
- t) A _____ is a digital or virtual currency secured by cryptography.
- u) _____ are blockchain-based "cryptographically unique tokens linked to digital or physical content, providing proof of ownership."
- v) _____ is by far the most widely used version control system.
- w) The "four Vs of big data are _____, _____, _____, and _____.
- x) Some people feel that the ultimate problem to be solved in AI is _____.
- y) ChatGPT is an example of the field of _____.

1.5

State which of the following are true and which are false. If false, explain why.

- a) Computing costs are dropping rapidly due to impressive developments in hardware, software and Internet technologies.
- b) For decades, computer power approximately doubled inexpensively every year. This remarkable trend is called Moore's law.
- c) Taking full advantage of multi-core architecture requires writing multithreaded applications.
- d) Any computer can directly understand only its own assembly language.
- e) The software that contains the core operating-system components is called the kernel.
- f) iOS is the most widely used mobile-phone operating system.
- g) C++ is the premiere language for building high-performance business-critical and mission-critical operating systems.
- h) The programming language most closely associated with the expression "write once, run anywhere" is Java.
- i) Swift is widely used to add programmability to web pages, such as animations and user interactivity.
- j) As the ARPANET evolved, to enable the huge variety of networking hardware and software to intercommunicate, ARPA developed the Internet Protocol (IP), which created a true "network of networks."
- k) Cloud computing allows you to increase or decrease computing resources to meet your needs at any given time.
- l) IoT is an example of edge computing.

- m) Augmented reality (AR) projects real-time computer-generated content onto what you see in the real world.
- n) Ethereum is a programmable platform that enables you to “build and deploy decentralized applications (dApps) on its network that use the blockchain to store data or control what your app can do.”
- o) Forbes defines Web3 as “the centralized Internet—built on distributed technologies like blockchain ... rather than on servers owned by individuals or corporations.”
- p) By some estimates, processing just one Bitcoin transaction uses approximately the same amount of energy as powering the average American home for one month.
- q) AI is often called a field with solutions but no problems.
- r) In machine learning, computers are explicitly programmed with human expertise to solve a particular problem.
- s) OpenAI’s Dall-E 2 can generate original images from text descriptions.

1.6 (*Clock as an Object*) Clocks are among the world’s most common objects. Discuss how each of the following terms and concepts applies to the notion of a clock: class, object, instantiation, data member, reuse, member function, inheritance (consider, for example, an alarm clock), base class and derived class.

1.7 (*Self-Driving Cars*) Just a few years ago, the notion of driverless vehicles on our streets would have seemed impossible. Many of the technologies you’ll read about in this book are making self-driving cars possible. Today, they’re already common in some areas.

- a) If you hailed a taxi and a driverless taxi stopped for you, would you get into the back seat? Would you feel comfortable telling it where you want to go and trusting it would get you there? What safety measures would you want in place?
- b) What if two self-driving cars approached a one-lane bridge from opposite directions? What protocol should they go through to determine which car should proceed?
- c) What if you’re behind a car stopped at a red light, the light turns green, and the car doesn’t move? You honk, and nothing happens. You get out of your car and notice that there’s no driver. What would you do?
- d) If a police officer pulls over a speeding self-driving car in which you’re the only passenger, who—or what entity—should pay the ticket?
- e) One especially serious concern with self-driving vehicles is that they might be hacked. What if the hacker sets the speed high (or low), which could be dangerous. What if the hacker locked all the doors then set the vehicle to drive off in the wrong direction?

1.8 (*Research: Artificial General Intelligence*) One of the most ambitious goals in the field of AI is to achieve artificial general intelligence—the point at which machine intelligence would equal human intelligence. Research this intriguing topic. When is this forecast to happen? What are some ethical issues this raises? Human intelligence seems to be stable over long periods. Powerful computers with artificial general intelligence could conceivably (and quickly) evolve intelligence far beyond that of humans. Research and discuss the issues this raises.

1.9 (Research: Intelligent Assistants) Many companies now offer computerized intelligent assistants, such as Amazon Alexa, Apple Siri, Google Assistant and Microsoft Cortana. Research these and others, and list uses that can improve people's lives. Research privacy and ethics issues for intelligent assistants. Research intelligent-assistant anecdotes. Would you do what an intelligent assistant told you to?

1.10 (Research: AI in Health Care) Research the rapidly growing field of AI big-data applications in health care. For example, suppose a diagnostic medical application had access to every x-ray that's ever been taken and the associated diagnoses—that's surely big data. "Deep Learning" computer-vision applications could work with this "labeled" data to learn to diagnose medical problems. Research deep learning in diagnostic medicine and list some of its most significant accomplishments. What are some ethical issues of having machines instead of human doctors performing medical diagnoses? Would you trust a machine-generated diagnosis? Would you seek a second opinion from another diagnostic machine? Would you insist on a second opinion from a human doctor?

1.11 (Research: Privacy and Data Integrity Legislation) Some key privacy laws are HIPAA (the Health Insurance Portability and Accountability Act) and the California Consumer Privacy Act (CCPA)—both in the United States, and GDPR (the General Data Protection Regulation) in the European Union. Laws like these are becoming more common and stricter. Investigate these laws and the major protections they provide for your privacy.

1.12 (Research: Personally Identifiable Information) Protecting users' personally identifiable information (PII) is an important aspect of privacy. Research and comment on this issue.

1.13 (Research: Big Data, AI and the Cloud—How Companies Use These Technologies) For a major organization of your choice, research how they might be using each of the following technologies: AI, big data, the cloud, mobile, natural-language processing, speech recognition, speech synthesis, database, machine learning, deep learning, reinforcement learning, Internet of Things (IoT) and web services.

1.14 (Research: Raspberry Pi and the Internet of Things) It's now possible to have a computer at the heart of just about any device and to connect those devices to the Internet. This has led to the Internet of Things (IoT), which is already connecting billions of devices. The Raspberry Pi is an economical computer often at the heart of IoT devices, and it can be programmed in C++. Research the Raspberry Pi and some of the many IoT applications in which it's used.

1.15 (Research: The Ethics of Deep Fakes) Artificial intelligence deep-learning technologies make it possible to create deep fakes—realistic fake videos of people that capture their appearance, voice, body motions and facial expressions. You can have them say and do whatever you specify. Research the ethics of deep fakes. What would happen if you turned on your TV and saw a deep-fake video of a prominent government official or newscaster reporting that an attack was about to happen? A famous low-tech version of this was Orson Welles's "War of the Worlds" radio broadcast of 1938, in which he created a mass panic by announcing that Martians were attacking New Jersey.

1.16 (Research: Blockchain—A World of Opportunity) Cryptocurrencies like Bitcoin and Ethereum are based on blockchain technology that has seen explosive growth over the

last few years. Research blockchain’s origin, applications and how it came to be used as the basis for cryptocurrencies. Research other major applications of blockchain. It is expected that blockchain technology will be crucial to implementing the Metaverse. Over the next many years, there will be extraordinary opportunities for software developers who thoroughly understand blockchain application development.

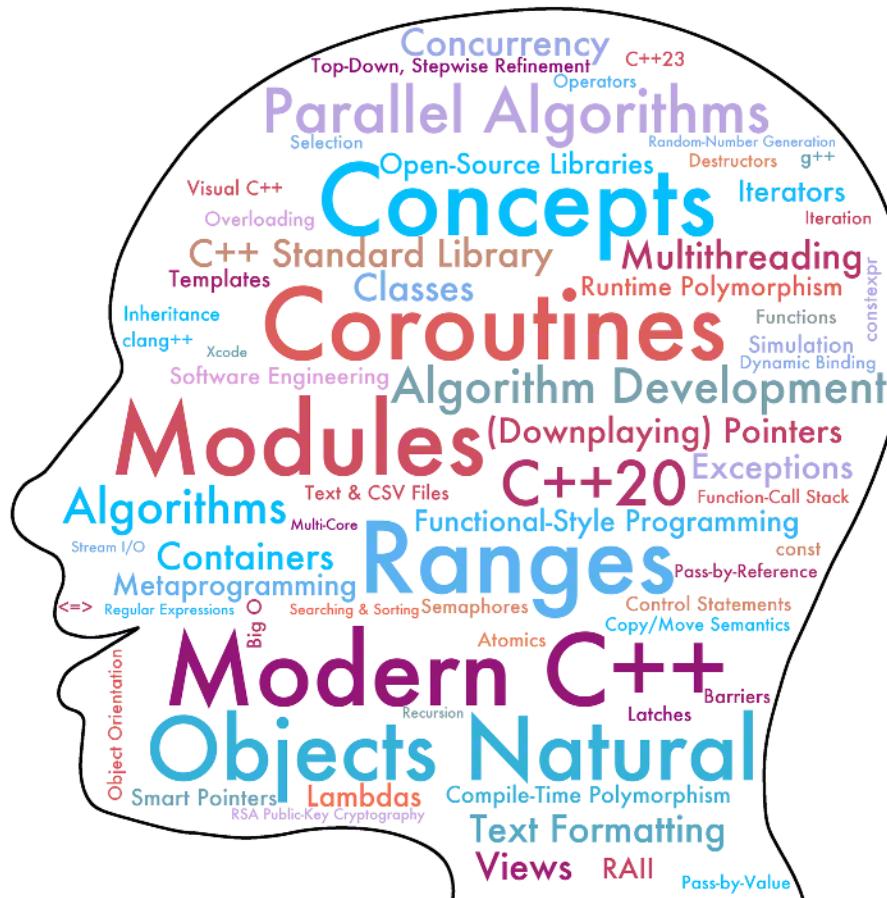
1.17 (Public-Key Cryptography) Cryptography is a crucial technology for privacy and security. In this book, you’ll have the opportunity to study both secret-key and public-key cryptography. Research how public-key cryptography is used to implement the Bitcoin cryptocurrency.

1.18 (Programmer Responsibility and Liability) As a programmer working in the health-care industry, suppose a software bug in one of your programs caused a cancer patient to receive an excessive dose of radiation therapy resulting in severe injury or death. Discuss the issues.

1.19 (2010 “Flash Crash”) An example of the consequences of our dependency on computers was the so-called “flash crash,” which occurred on May 6, 2010, when the U.S. stock market fell precipitously in a matter of minutes, wiping out trillions of dollars of investments—then recovered within minutes. Use the Internet to investigate the causes of this crash and discuss the issues it raises.

2

Intro to C++20 Programming



Objectives

In this chapter, you'll:

- Write simple C++ applications.
- Use input and output statements.
- Use fundamental data types.
- Understand basic memory concepts.
- 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 conveniently creating and using powerful objects of the C++ standard library’s `string` class even before you learn to code your own custom classes in later chapters.

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 Memory Concepts
2.6 Arithmetic | 2.7 Decision Making: Equality and Relational Operators
2.8 Objects Natural Case Study: Creating and Using Objects of Standard-Library Class <code>string</code>
2.9 Wrap-Up Exercises |
|---|--|

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 examples output messages on the screen. The fourth example 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 how to compare two numbers, then display messages based on the comparison results.

The “Objects Natural” Learning Approach

In your programs, you’ll create and use many objects of preexisting carefully-developed-and-tested classes that enable you to perform significant tasks with minimal code, even in the book’s earliest chapters. 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 created by the massive open-source communities that have developed around key programming languages.

To help you appreciate this programming style even in the book’s introductory chapters, you’ll conveniently create and use objects of preexisting C++ standard library classes without yet knowing how to create 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. Beginning in Chapter 9, you’ll craft valuable custom classes for your own use and for reuse by other programmers.

Compiling and Running Programs

For instructions on compiling and running programs, see the test-drives in Chapter 1.

2.2 First Program in C++: Displaying a Line of Text

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

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

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

Fig. 2.1 | Text-printing program.

Comments

Lines 1 and 2

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

both begin with `//`, indicating that the remainder of each line is a **comment**. You insert comments to document your programs and help other people read and understand them. Comments do not cause the computer to perform any action when the program is run—they’re ignored by the C++ compiler. In our programs, the first-line comment contains the program’s file name. The second-line comment “Text-printing program.” describes the program’s purpose. A comment beginning with `//` is called a **single-line comment** because it terminates at the end of the current line. You can create single or **multiline comments** by enclosing them in `/*` and `*/`, as in

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

or

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

#include Preprocessing Directive

Line 3

```
#include <iostream> // enables program to output data to the screen
```

is a **preprocessing directive**—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 19 explains the contents of `<iostream>` in more detail. Forgetting to include `<iostream>` in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message. 

Blank Lines and Whitespace

Line 4 is a blank line. Blank lines, spaces and tabs make programs easier to read. Together, these characters are known as **whitespace**—they’re usually 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. One function in every program must be named `main`. This is where C++ programs begin executing. The keyword `int` indicates that after `main` finishes executing, it “returns” an integer (whole number) value. **Keywords**, like `return`, are reserved for use by C++. 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.

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

An Output Statement

Line 7

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

displays the characters contained between the double quotation marks. 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. Whitespace 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. 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 usually “connected” to the screen.



Omitting the semicolon at the end of a C++ statement is a syntax error. The **syntax** of a programming language specifies the rules for creating proper programs in that language. A **syntax error** occurs when the compiler encounters code that violates C++’s language rules (i.e., its syntax). The compiler issues an error message to help you locate and fix the incorrect code. Syntax errors are also called **compiler errors**, **compile-time errors** or **compilation errors** because the compiler detects them during the compilation phase. You cannot execute your program until you correct all the syntax errors. As you’ll see, some compilation errors are not syntax errors.

Indentation

Indent each function’s body one level within the braces that delimit the body. This makes a program’s functional structure stand out, making the program easier to read. Set a convention for the indent size 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 from standard-library headers like `<iostream>`. The notation `std::cout` specifies that we are using the name `cout`, which belongs to **namespace std**.¹ We discuss namespaces in Chapter 16. For now, include `std::` before each mention of `cout`. 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

When used with `cout`, the `<<` operator is the **stream insertion operator**. The value to the operator's right (the right **operand**) is inserted in the output stream. Note that `<<` points toward where the data goes. A string's characters are typically displayed exactly as typed between the quotes. However, the characters `\n` are *not* displayed in Fig. 2.1's output. The backslash (`\`) is an **escape character**. When C++ encounters a backslash in a string, it combines the next character with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. It causes the **cursor**—that is, the current screen-position indicator—to move to the beginning of the next line on the screen. Some common escape sequences are shown in the following table:

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

The return Statement

Line 9

```
return 0; // indicate that program ended 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. The **main** function implicitly returns 0 if program execution reaches the closing brace without encountering a **return statement**. For this reason, we omit `main`'s `return` statement in subsequent programs that terminate successfully.



Checkpoint

| (Fill-in) Together, blank lines, spaces and tabs are known as _____ and are usually ignored by the compiler.

Answer: whitespace.

1. We pronounce “`std::`” as “standard,” rather as its individual letters `s`, `t` and `d`.

- 2 (Code)** Write a statement that outputs the string "Hello world" then moves the cursor to the beginning of the next line.

Answer: `std::cout << "Hello World\n"; // display message`

- 3 (True/False)** The following statement, when executed at the end of `main`, indicates that the program did not execute correctly:

```
return 0;
```

Answer: False. This statement indicates that the program executed 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. This string did not end with `\n`, so line 8 begins displaying on the same line immediately following the space.

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



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

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

Displaying Multiple Lines of Text with a Single Statement

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

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

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

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

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



Checkpoint

- 1 (*True/False*) The following statement displays the string "Hello" then positions the output cursor on the next line.

```
std::cout << "Hello";
```

Answer: False. This statement displays the string "Hello" and leaves the output cursor positioned after the "o" in "Hello".

- 2 (*True/False*) A single statement can display multiple lines by using additional newline characters.

Answer: True.

- 3 (*Code*) Write a statement that outputs a single string, displaying the following and leaves the output cursor positioned on a new line:

```
Hello
```

```
World
```

Answer: std::cout << "Hello\n\nWorld\n";

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 sample executions, our convention is to show the user's input in **bold**.

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

Fig. 2.4 | Addition program that displays the sum of two integers. (Part 1 of 2.)

```

17     sum = number1 + number2; // add the numbers; store result in sum
18
19     std::cout << "Sum is " << sum << "\n"; // display sum
20 }
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. (Part 2 of 2.)

Variable Declarations and Braced Initialization

Lines 8–10

```

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)

```

are **declarations**, and `number1`, `number2` and `sum` are the names of **variables**. A variable is a location in the computer’s memory where a value can be stored for use by a program. These declarations specify that the variables `number1`, `number2` and `sum` are data of type **int**, meaning they 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.

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 **braced 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.

Lines 8–10 also can be written as:

```

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)

```

In legacy C++ programs, you’re likely to encounter initialization statements using this older coding style. Our code examples use braced initialization. In subsequent chapters, we’ll discuss various benefits of braced initializers.

Declaring Multiple Variables at Once

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

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

However, 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 `char` for character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and -11.19. A `char` variable may hold one lowercase letter, uppercase letter, digit or special character (e.g., \$ or *). Types such as `int`, `double`, `char` and `long` are called **fundamental types** and are built into

C++. Fundamental-type names typically consist of one or more keywords and must appear in all lowercase letters. For a complete list of C++ fundamental types and their ranges, see

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

Identifiers and Camel-Case Naming

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

C++ allows identifiers of any length. Do not begin an identifier with an underscore and a capital letter or two underscores—C++ compilers use such names for their own purposes internally.

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter. For example, `firstNumber` starts its second word, `Number`, with a capital N. This naming convention is known as **camel case** because the uppercase letters stand out like a camel's humps. Choosing meaningful identifiers helps make a program **self-documenting**—a person can understand the program simply by reading it rather than referring to program comments or external documentation. Avoid using abbreviations in identifiers. Although this may require you to type more keystrokes, it improves program readability.

Placement of Variable Declarations

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

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

could have been placed immediately before line 13:

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

the declaration in line 9:

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

could have been placed immediately before line 16:

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

and the declaration in line 10:

```
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

could have been placed immediately before line 18:

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

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

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

Obtaining the First Value from the User

Line 12

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

displays `Enter first integer:` followed by a space. This message is called a **prompt** because it directs the user to take a specific action. Line 13

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

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 program waits for you to enter a value for variable `number1`. You respond by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the program. The `cin` object 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 expects 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). In these early programs, we assume that the user enters valid data. We'll present various techniques for dealing with data-entry problems later.

Obtaining the Second Value from the User

Line 15

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

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

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

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

Calculating the Sum of the Values Input by the User

The assignment statement in line 18

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

adds the values of `number1` and `number2` and assigns the result to `sum` using the **assignment operator `=`**. Most calculations are performed in assignment statements. The `=` operator and the `+` operator are **binary operators**—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 a space on each side of a binary operator makes the operator stand out and makes the program more readable.

Displaying the Result

Line 20

```
std::cout << "Sum is " << sum << "\n"; // display sum
```

displays the string `"Sum is"` followed by the numerical value of variable `sum` and a newline. This statement outputs values of multiple types. The stream insertion operator “knows” how to output each type of data. Using multiple stream insertion operators (`<<`) in a single statement is called **concatenating, chaining** or **cascading stream insertion operations**.

Calculations can also be performed in output statements. We could have eliminated the variable `sum` by combining the statements in lines 18 and 20 into the statement

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

A signature feature of C++ is that you can create your own data types called classes, which we discuss in depth in Chapter 9 and subsequent chapters. 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 11.



Checkpoint

1 (Code) Write a statement that initializes the `int` variable `total` to 0 using braced initialization.

Answer: `int total{0};`

2 (Code) Assume the `int` variables `billAmount` and `tip` exist and have been initialized. Write an initialization statement that adds the values of `billAmount` and `tip` and stores the result in the new `int` variable `totalBill`.

Answer: `int totalBill{billAmount + tip};`

3 (True/False) Variables must be defined with a type and a name before using them.

Answer: True.

2.5 Memory Concepts

Variable names such as `number1`, `number2` and `sum` correspond to **locations** in the computer’s memory. Every variable has a **name**, a **type**, a **size** (determined by the type) and a **value**.

In the addition program of Fig. 2.4, when the following statement (line 13) executes

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

the program places the integer typed by the user into a memory location to which the compiler assigned the name `number1`. If the user enters 45 for `number1`, the program places 45 into the location `number1`, as in:

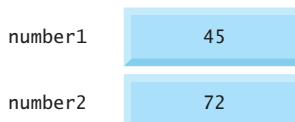


When a value is placed in a memory location, the new value overwrites the previous one in that location—that previous value is lost.

Returning to our addition program, suppose the user enters 72 when the following statement (line 16) executes:

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

The program places 72 into the location `number2`, and memory appears as in:

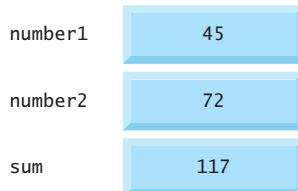


The variables’ locations are not necessarily adjacent in memory.

After the program obtains values for `number1` and `number2`, it adds these values and places the total into the variable `sum`. The statement (line 18)

```
sum = number1 + number2;
```

places the calculated sum of `number1` and `number2` into variable `sum`, after which memory appears as follows:



Performing the calculation uses, but does not modify, the values of `number1` and `number2`.



Checkpoint

1 (*Fill-in*) Every variable has a name, a _____, a _____ and a _____.

Answer: type, size, value.

2 (*True/False*) Storing a value in a variable replaces the previous value in the corresponding memory location.

Answer: True.

3 (*True/False*) An expression like `number1 + number2` uses the values of the variables `number1` and `number2`, but does not modify them.

Answer: True.

2.6 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>

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. So, the expression `7 / 4` evaluates to 1, and the expression `17 / 5` evaluates to 3. Any fractional part resulting from integer division is truncated—no rounding occurs.

Remainder Operator

The **remainder operator**, % (also called the **modulus operator**), yields the remainder after integer division and can be used only with integer operands. The expression `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 x times the quantity $y + z$, we write $x * (y + z)$.

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**, as in the following statement, expressions in the innermost pair of parentheses evaluate first:

$$(a * (b + c))$$
2. Multiplication, division and remainder operations evaluate next. In an expression containing several of these operations, they’re applied from left to right. These three operators have the same level of precedence.
3. Addition and subtraction operations evaluate last and are applied from left to right. Addition and subtraction have the same level of precedence.

You can view the complete operator precedence chart at https://en.cppreference.com/w/cpp/language/operator_precedence. **Caution:** In an expression with two sets of parentheses “on the same level,” such as such as $(a + b) * (c - d)$, 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** (sometimes called **associativity**). 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 from left to right. We’ll see that some operators group from right to left.

Sample Algebraic and C++ Expressions

Now consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent. The following is an example of an arithmetic mean (average) of five terms:

$$\text{Algebra: } m = \frac{a + b + c + d + e}{5}$$

$$\text{C++: } m = (a + b + c + d + e) / 5;$$

The parentheses are required because division has higher precedence than addition. The *entire* quantity $(a + b + c + d + e)$ is to be divided by 5. If the parentheses are erroneously omitted, we obtain $a + b + c + d + e / 5$, which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

$$\text{Algebra: } y = mx + b$$

$$\text{C++: } y = m * x + b;$$



No parentheses are required. Multiplication is applied first because it has higher precedence than addition.

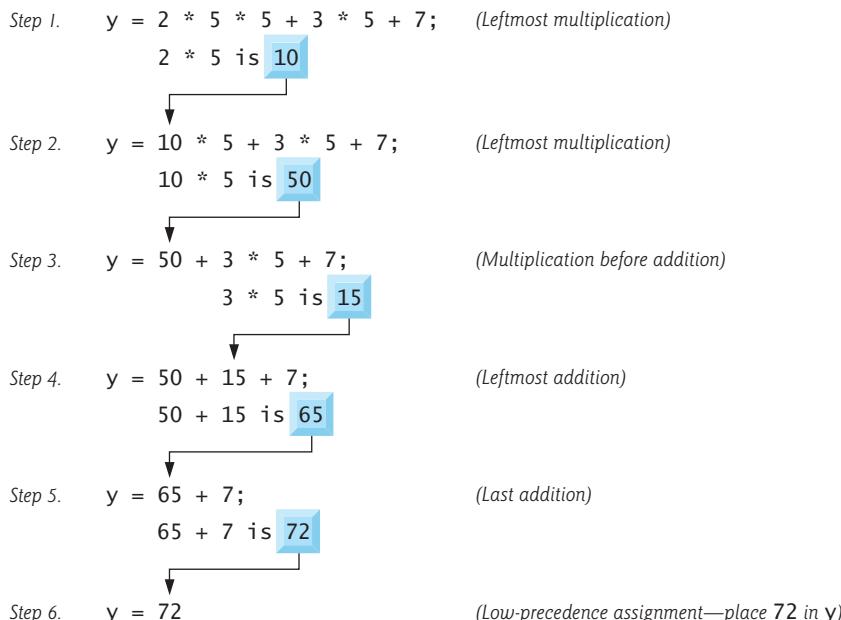
Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial $y = ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

There is no arithmetic operator for exponentiation in C++, so we've represented x^2 as $x * x$. The circled numbers under the statement indicate the order in which C++ applies the operators. The assignment operator is applied last because its precedence is lowest among the operators in the statement.

Suppose variables a , b , c and x in the preceding second-degree polynomial are initialized as follows: $a = 2$, $b = 3$, $c = 7$ and $x = 5$. The following diagram illustrates the order in which the operators are applied and the final value of the expression:



Redundant Parentheses

As in algebra, it's acceptable to place unnecessary parentheses in an expression to make it clearer. These are called **redundant parentheses**. For example, the second-degree polynomial could be parenthesized as follows:

```
y = (a * x * x) + (b * x) + c;
```



Checkpoint

1 (*True/False*) The expression `10 / 4`, which contains two integer operands, produces the quotient `2.5`.

Answer: False. Integer division produces an integer quotient, so the quotient is `2`.

2 (*Fill-in*) The result of the expression `10 % 3` is _____, and the result of the expression `10 % 7` is _____.

Answer: `1, 3`.

3 (*Code*) Write an expression that divides `number1` by the difference between `number2` and `number3`.

Answer: `number1 / (number2 - number3)`

2.7 Decision Making: Equality and Relational Operators

We now introduce C++'s **if statement**, which allows a program to perform different actions 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>			
<code>></code>	<code>></code>	<code>x > y</code>	<code>x</code> is greater than <code>y</code>
<code><</code>	<code><</code>	<code>x < y</code>	<code>x</code> is less than <code>y</code>
<code>≥</code>	<code>≥</code>	<code>x ≥ y</code>	<code>x</code> is greater than or equal to <code>y</code>
<code>≤</code>	<code>≤</code>	<code>x ≤ y</code>	<code>x</code> is less than or equal to <code>y</code>
<i>Equality operators</i>			
<code>=</code>	<code>==</code>	<code>x == y</code>	<code>x</code> is equal to <code>y</code>
<code>≠</code>	<code>!=</code>	<code>x != y</code>	<code>x</code> is not equal to <code>y</code>

The relational operators all have the same precedence and group from left-to-right. The equality operators have the same precedence—lower than that of the relational operators—and group from left-to-right.

Reversing the order of the symbols in the operators `!=`, `≥` and `≤` (by writing them as `!=`, `≥>` and `≤<`, respectively) is usually a syntax error. In some cases, writing `!=as=!` will not be a syntax error but almost certainly will be a runtime **logic error**—an error that causes the program to execute incorrectly. You'll understand why when we cover logical operators in Section 4.11.



Err

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.” Confusing these operators may not necessarily cause an easy-to-recognize syntax error, but it may cause subtle logic errors. Compilers generally warn about this.



Err

Using the if Statement

The program of Fig. 2.5 (followed by several executions) 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 is skipped.

```

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
9 // function main begins program execution
10 int main() {
11     int number1{0}; // first integer to compare (initialized to 0)
12     int number2{0}; // second integer to compare (initialized to 0)
13
14     cout << "Enter two integers to compare: "; // prompt user for data
15     cin >> number1 >> number2; // read two integers from user
16
17     if (number1 == number2) {
18         cout << number1 << " == " << number2 << "\n";
19     }
20
21     if (number1 != number2) {
22         cout << number1 << " != " << number2 << "\n";
23     }
24
25     if (number1 < number2) {
26         cout << number1 << " < " << number2 << "\n";
27     }
28
29     if (number1 > number2) {
30         cout << number1 << " > " << number2 << "\n";
31     }
32
33     if (number1 <= number2) {
34         cout << number1 << " <= " << number2 << "\n";
35     }
36
37     if (number1 >= number2) {
38         cout << number1 << " >= " << number2 << "\n";
39     }
40 } // end function main

```

```

Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7

```

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

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

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

using Declarations

Lines 6–7

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

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

using Directive

In place of lines 6–7, many programmers prefer the **using directive**

```
using namespace std;
```

which enables your program to use names from the `std` namespace without the `std::` qualification. In the early chapters, we'll use this directive to simplify the code,² but in Chapter 6 we'll return to using `std::`, which is considered good practice and can help you avoid some subtle coding errors.



Variable Declarations and Reading the Inputs from the User

Lines 11–12

```
int number1{0}; // first integer to compare (initialized to 0)
int number2{0}; // second integer to compare (initialized to 0)
```

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

Line 15

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

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. This statement first reads a value into `number1`, then reads another value into `number2`.

2. In Chapter 20, we'll discuss some disadvantages of **using** directives in large-scale systems.

Comparing Numbers

The `if` statement in lines 17–19

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

determines whether the values of variables `number1` and `number2` are equal. If so, the output statement (line 18) 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 21, 25, 29, 33 and 37, the corresponding output 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 such errors, as a rule, always enclose an `if` statement's body statement(s) in braces. Also, indent the statement(s) in an `if` statement's block to enhance readability.



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. All three of our preferred compilers issue a warning for this logic error.

Splitting Lengthy Statements

A lengthy statement may be spread over several lines. If you must do this, 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, it's a good practice to indent all subsequent lines.

Operator Precedence and Grouping

Except for 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 to `x` the result of that assignment—that is, the value of `y`, which is now 0.

Refer to the complete operator-precedence chart at https://en.cppreference.com/w/cpp/language/operator_precedence 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 a sequence of smaller statements or use parentheses to force the order of evaluation, just as you'd do in an algebraic expression.



Checkpoint

- 1 *(Code)* Write a `using` directive that enables you to use `cin` and `cout` without the `std::` qualification.

Answer: `using namespace std;`

- 2 *(Code)* Assume `number1` is 12 and `number2` is 23. What does the following code print?

```
if (number1 > number2) {  
    cout << number1 << " > " << number2 << "\n";  
}
```

Answer: The statement will not print anything because the condition is false, so the body of the statement will be skipped.

- 3 *(Code)* Assume `number1` is 23 and `number2` is 23. What does the following code print? [Write and execute a short program to see what this code actually does.]

```
if (number1 != number2); {  
    cout << number1 << " != " << number2 << "\n";  
}
```

Answer: This code will incorrectly print `23 != 23` because a semicolon is incorrectly placed immediately following the `if` statement's condition.

2.8 Objects Natural Case Study: Creating and Using Objects of Standard-Library Class `string`

Throughout this book, we emphasize using valuable preexisting classes from the C++ standard library and various 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 conveniently create and program with powerful objects even before you learn to create your own custom classes in Chapter 9.

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`³ class.

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` objects.

3. You'll learn additional `string` capabilities in subsequent chapters. Chapter 8 discusses class `string` in detail, test-driving many more of its member functions.

```

1 // fig02_06.cpp
2 // Standard library string class test program.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main() {
8     string s1{"happy"};
9     string s2{" birthday"};
10    string s3; // creates an empty string
11
12    // display the strings and show their lengths
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") << "\n";
33    cout << "s2 starts with \"ha\": " << s2.starts_with("ha") << "\n";
34    cout << "s1 ends with \"ay\": " << s1.ends_with("ay") << "\n";
35    cout << "s2 ends with \"ay\": " << s2.ends_with("ay") << "\n";
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 class⁴—also called instantiating an object. Lines 8–10 create three `string` objects:

- `s1` is initialized with a copy of the string literal "happy",
- `s2` is initialized with a copy of the string literal " birthday", and
- `s3` is initialized by default to the **empty string** (that is, "").

When we declare `int` variables, as we did earlier, the compiler knows what `int` is—it's a fundamental type 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 languages like C++ that support object-oriented programming. Such languages 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`.

`string` Member Function `length`

Lines 13–15 output each `string` and its length. The `string` class's **length member function** 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* parentheses indicate that `length` does not require 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 how `length` performs its task, just as the driver of a car doesn't know 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.

4. You'll see in Section 9.20 that you can call a class's `static` member functions without creating an object of that class.

Comparing string Objects with the Equality Operators

Like numbers, strings can be compared with one another. Lines 18–20 compare `s2` to `s1` using the equality operators—string comparisons are case sensitive.⁵

Usually, 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 more naturally 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—that is, the length of the `string` is 0. Otherwise, `empty` 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—known as `string concatenation`. After the assignment, `s3` contains the characters of `s1` followed by the characters of `s2`—“happy birthday”. Line 28 outputs `s3` to demonstrate that the assignment worked correctly.

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

Lines 32–35 demonstrate C++20 `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.



Checkpoint

- 1** *(Code)* Assume the `string` object `name` exists and is initialized. Write a `string` member-function call that returns `true` if the `name` starts with “Je”.

Answer: `name.starts_with("Je")`

- 2** *(Code)* Assume the `string` object `name` exists and is initialized. Write a `string` member-function call that returns `true` if the `name` ends with “on”.

Answer: `name.ends_with("on")`

2.9 Wrap-Up

We presented many important C++ features in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. 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 (also called the associativity 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.

5. In Chapter 8, you'll see that strings perform “lexicographical” comparisons using the numerical values of the characters in each string.

Finally, we introduced our “Objects Natural” approach to learning C++ 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 9–11, 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 to be performed.

Exercises

- 2.1** Fill in the blanks in each of the following:
- _____ are used to document a program and improve its readability.
 - The object used to print information on the screen is _____.
 - A C++ statement that makes a decision is _____.
 - Most calculations are normally performed by _____ statements.
 - The _____ object inputs values from the keyboard.
 - What arithmetic operations are on the same level of precedence as multiplication? _____.
 - When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
 - A location in the computer’s memory that may contain different values throughout the execution of a program is called a(n) _____.
- 2.2** Discuss the meaning of each of the following objects:
- `std::cin`
 - `std::cout`
- 2.3** State whether each of the following is *true* or *false*. If *false*, explain why. Assume the statement `using std::cout;` is used.
- Comments cause the computer to print the text after the // on the screen when the program is executed.
 - Displaying the escape sequence \n causes the cursor to position to the beginning of the next line on the screen.
 - All variables must be declared before they’re used.
 - All variables must be given a type when they’re declared.
 - C++ considers the variables `number` and `NuMbEr` to be identical.
 - Declarations can appear almost anywhere in the body of a C++ function.
 - The remainder operator (%) can be used only with integer operands.
 - The arithmetic operators *, /, %, + and – all have the same level of precedence.
 - All operators are evaluated from left to right.
 - The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
 - The statement `cout << "a = 5;"`; is an assignment statement.
 - A valid arithmetic expression with no parentheses is evaluated from left to right.
 - The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.
- 2.4** Write a single C++ statement to accomplish each of the following (assume that neither `using` declarations nor a `using` directive has been used):

- a) Declare the variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int` (in one statement) and initialize each to 0.
- b) Prompt the user to enter an integer. End your prompting message with a colon (`:`) followed by a space, and leave the cursor positioned after the space.
- c) Read an integer from the user at the keyboard and store it in `int` variable `age`.
- d) If the variable `number` is not equal to 7, print "The variable `number` is not equal to 7".
- e) Print "This is a C++ program" on one line.
- f) Print "This is a C++ program" on two lines. End the first line with `C++`.
- g) Print "This is a C++ program" with each word on a separate line.
- h) Print "Enter two numbers".
- i) Assign the product of variables `b` and `c` to variable `a`.
- j) State that a program performs a payroll calculation (i.e., use text that helps to document a program).
- k) Input three integer values from the keyboard into integer variables `a`, `b` and `c`.

2.5 Identify and correct the errors in each of the following statements (assume that the statement `using std::cout;` is used):

- a) `if (c < 7) {`
 `cout << "c is less than 7\n";`
`}`
- b) `if (c => 7) {`
 `cout << "c is equal to or greater than 7\n";`
`}`

2.6 What, if anything, prints when each of the following statements executes? If nothing prints, answer "nothing." Assume `x = 2` and `y = 3`.

- a) `cout << x;`
- b) `cout << x + x;`
- c) `cout << "x=";`
- d) `cout << "x = " << x;`
- e) `cout << x + y << " = " << y + x;`
- f) `z = x + y;`
- g) `cin >> x >> y;`
- h) `// cout << "x + y = " << x + y;`
- i) `cout << "\n";`

2.7 Which of the following statements contain variables whose values are replaced?

- a) `cin >> b >> c >> d >> e >> f;`
- b) `p = i + j + k + 7;`
- c) `cout << "variables whose values are replaced";`
- d) `cout << "a = 5";`

2.8 Given the algebraic equation $y = ax^3 + 7$, which of the following, if any, are correct C++ statements for this equation?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`

- e) $y = a * (x * x * x) + 7;$
 f) $y = a * x * (x * x + 7);$

2.9 (Order of Evaluation) State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

- a) $x = 7 + 3 * 6 / 2 - 1;$
 b) $x = 2 \% 2 + 2 * 2 - 2 / 2;$
 c) $x = (3 * 9 * (3 + (9 * 3 / (3))));$

2.10 Write a statement (or comment) to accomplish each of the following (assume that using declarations have been used for `cin`, `cout` and `endl`):

- Document that a program calculates the product of three integers.
- Declare the variables x , y , z and `result` to be of type `int` (in separate statements) and initialize each to 0.
- Prompt the user to enter three integers.
- Read three integers from the keyboard and store them in variables x , y and z .
- Compute the product of the three integers in variables x , y and z , and assign the result to the variable `result`.
- Print "The product is " followed by the value of the variable `result`.

2.11 Using the statements you wrote in Exercise 2.10, write a complete program that calculates and displays the product of three integers. Add comments to the code where appropriate. [Note: You'll need to write the necessary `using` declarations or directive.]

2.12 (Arithmetic) Write a program that asks the user to enter two integers, obtains the numbers from the user and prints their sum, product, difference, and quotient.

2.13 (Printing) Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Do this in several ways:

- Using one statement with one stream insertion operator.
- Using one statement with four stream insertion operators.
- Using four statements.

2.14 (Comparing Integers) Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "is larger." If the numbers are equal, print the message "These numbers are equal."

2.15 (Arithmetic, Smallest and Largest) Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialog should appear as follows:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.16 (Diameter, Circumference and Area of a Circle) Write a program that reads a circle's radius as an integer and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π . Do all calculations in output statements. [Note: In this chapter, we've discussed only integer constants and variables. In Chapter 4, we discuss floating-point numbers, i.e., values that have decimal points.]

2.17 What does the following code print?

```
cout << "*\n**\n***\n****\n*****\n";
```

2.18 (*Largest and Smallest Integers*) Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.19 (*Odd or Even*) Write a program that reads an integer and determines and prints whether it's odd or even. [Hint: Use the remainder operator (%). An even number is a multiple of two. Any multiple of 2 leaves a remainder of zero when divided by 2.]

2.20 (*Multiples*) Write a program that reads in two integers and determines and prints if the first is a multiple of the second. [Hint: Use the remainder operator (%).]

2.21 (*Checkerboard Pattern*) Display the following checkerboard pattern with eight output statements, then display the same pattern using as few statements as possible.

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

2.22 (*Integer Equivalent of a Character*) Here is a peek ahead. In this chapter, you learned about integers and the type `int`. C++ can also represent uppercase letters, lowercase letters and various special symbols. C++ uses small integers internally to represent each different character. The set of characters a computer uses and the corresponding integer representations for those characters are called that computer's **character set**. You can print a character by enclosing that character in single quotes, as with

```
cout << 'A'; // print an uppercase A
```

You can print the integer equivalent of a character using `static_cast` as follows:

```
cout << static_cast<int>('A'); // print 'A' as an integer
```

This is called a **cast** operation. The expression `static_cast<int>('A')` converts the character 'A' to its integer equivalent 65 (on systems that use the ASCII character set or Unicode character set). Write a program that prints the integer equivalent of a character the user enters at the keyboard. Store the input in a variable of type `char`. Test your program several times using uppercase letters, lowercase letters, digits and special characters (such as \$).

2.23 (*Digits of an Integer*) Write a program that inputs a five-digit integer, separates the integer into its digits and prints them with three spaces between each. For example, if the user types in 42339, the program should print:

```
4   2   3   3   9
```

[Hint: Use the integer division and remainder operators.]

2.24 (*Table*) Using only the techniques of this chapter, write a program that calculates the squares and cubes of the integers from 0 to 10. Use tabs to print the following neatly formatted table of values:

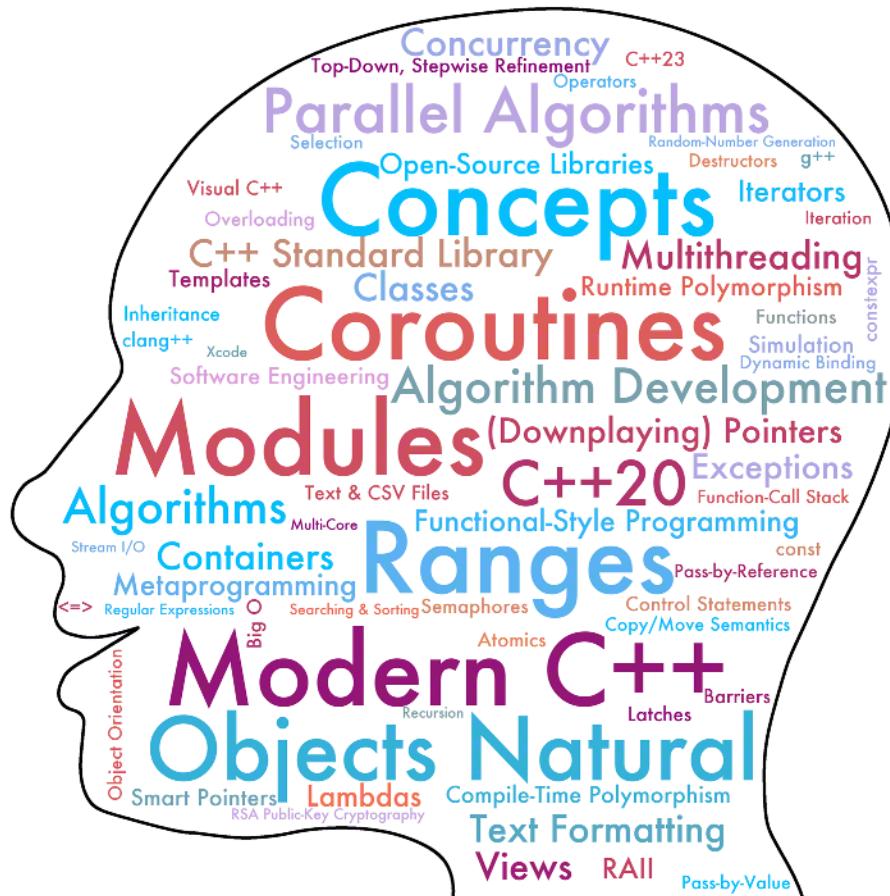
integer	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

In later chapters, you'll learn more elegant ways to solve this problem.

This page intentionally left blank

3

Algorithm Development and Control Statements: Part I



Objectives

In this chapter, you'll:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- 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 our Objects Natural approach with a case study on creating and manipulating integers as large as you want them to be.

Outline

3.1	Introduction	
3.2	Algorithms	
3.3	Pseudocode	
3.4	Control Structures	
3.4.1	Sequence Structure	
3.4.2	Selection Statements	
3.4.3	Iteration Statements	
3.4.4	Summary of Control Statements	
3.5	if Single-Selection Statement	
3.6	if...else Double-Selection Statement	
3.6.1	Nested if...else Statements	
3.6.2	Blocks	
3.6.3	Conditional Operator (? :)	
3.7	while Iteration Statement	
3.8	Formulating Algorithms: Counter-Controlled Iteration	
3.8.1	Pseudocode Algorithm with Counter-Controlled Iteration	
3.8.2	Implementing Counter-Controlled Iteration	
3.8.3	Integer Division and Truncation	
3.8.4	Arithmetic Overflow	
3.8.5	Input Validation	
3.9	Formulating Algorithms: Sentinel-Controlled Iteration	
3.9.1	Top-Down, Stepwise Refinement: The Top and First Refinement	
3.9.2	Proceeding to the Second Refinement	
3.9.3	Implementing Sentinel-Controlled Iteration	
3.9.4	Mixed-Type Expressions and Implicit Type Promotions	
3.9.5	Formatting Floating-Point Numbers	
3.10	Formulating Algorithms: Nested Control Statements	
3.10.1	Problem Statement	
3.10.2	Top-Down, Stepwise Refinement: Pseudocode Representation of the Top	
3.10.3	Top-Down, Stepwise Refinement: First Refinement	
3.10.4	Top-Down, Stepwise Refinement: Second Refinement	
3.10.5	Complete Second Refinement of the Pseudocode	
3.10.6	Implementing the Program	
3.10.7	Preventing Narrowing Conversions with Braced Initialization	
3.11	Compound Assignment Operators	
3.12	Increment and Decrement Operators	
3.13	Fundamental Types Are Not Portable	
3.14	Objects Natural Case Study: Super-Sized Integers	
3.15	Wrap-Up Exercises	

3.1 Introduction

Before writing a program to solve a problem, you should have a thorough understanding of it and a carefully planned approach to solving it. When writing a program, you also should understand the available building blocks and employ proven program-construction techniques. In this chapter and the next, we present the theory and principles of structured programming. As we'll see in later chapters, the concepts presented here will be crucial in building classes and manipulating objects.

We discuss the **if** statement in additional detail and introduce the **if...else** and **while** statements—all of these building blocks allow you to specify the logic required for functions to perform their tasks. We also introduce the compound assignment operators and the increment and decrement operators. We discuss why the fundamental types are not portable. We continue our Objects Natural approach with a case study on creating and manipulating super-sized integers that can represent values beyond the ranges of integers supported by computer hardware.

3.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. An **algorithm** is a procedure for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which these actions execute

The following example demonstrates that correctly specifying the order in which the actions execute is essential.

Consider the “rise-and-shine algorithm” one executive follows for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a program is called **program control**. This chapter investigates program control using C++’s **control statements**.



Checkpoint

1 (*Fill-in*) Specifying the order in which statements (actions) execute in a program is called _____.

Answer: program control.

2 (*Fill-in*) A procedure for solving a problem in terms of the _____ to execute and the _____ in which these actions execute is called an algorithm.

Answer: actions, order.

3.3 Pseudocode

Pseudocode is an informal language that helps you develop algorithms without worrying about the strict details of C++ language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of C++ programs. Pseudocode is similar to everyday English (or your preferred language)—it’s convenient and user-friendly, but it’s not an actual computer programming language. You’ll see an algorithm written in pseudocode momentarily.

Pseudocode does not execute on computers. Rather, it helps you “think out” a program before attempting to write it in a programming language like C++. This chapter provides several examples of using pseudocode to develop C++ programs.

The style of pseudocode we present consists purely of characters, so you can type pseudocode conveniently using any text-editor program. A carefully prepared pseudocode program can easily be converted to a corresponding C++ program.

Pseudocode typically describes only statements representing the actions that occur after you convert a program from pseudocode to C++ and run the program. Such actions might include input, output, assignments or calculations. In our pseudocode, we typically do not include variable declarations, but some programmers choose to list variables and mention their purposes.

Addition Program Pseudocode

Let's look at a pseudocode example that might help a programmer create the addition program of Fig. 2.4. The following pseudocode corresponds to the algorithm that inputs two integers from the user, adds them and displays their sum. We show the complete pseudocode listing here. We'll show how to create pseudocode from a problem statement later in the chapter.

-
- 1 Prompt the user to enter the first integer
 - 2 Input the first integer
 - 3
 - 4 Prompt the user to enter the second integer
 - 5 Input the second integer
 - 6
 - 7 Add first integer and second integer, store result
 - 8 Display result
-

The pseudocode statements are simply English statements that convey what task is to be performed in C++. Lines 1–2 correspond to the C++ statements in lines 12–13 of Fig. 2.4. Lines 4–5 correspond to the statements in lines 15–16, and lines 7–8 correspond to the statements in lines 18 and 20.



Checkpoint

- 1 (Fill-in) The actions described by pseudocode might include _____, _____, _____ or _____.

Answer: input, output, assignments, calculations.

- 2 (Fill-in) _____ is an informal language that helps you develop algorithms without worrying about the strict details of C++ language syntax.

Answer: Pseudocode.

3.4 Control Structures

Typically, program statements execute one after the other in the order in which they're written. This process is called sequential execution. Various C++ statements, which we'll soon discuss, enable you to specify that the next statement to execute is not necessarily the next one in sequence. This is called transfer of control.

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 Böhm and Jacopini¹ demonstrated that programs could be written without any **goto** statements. The challenge for the era's programmers was shifting their styles to "goto-less programming." The term **structured programming** became almost

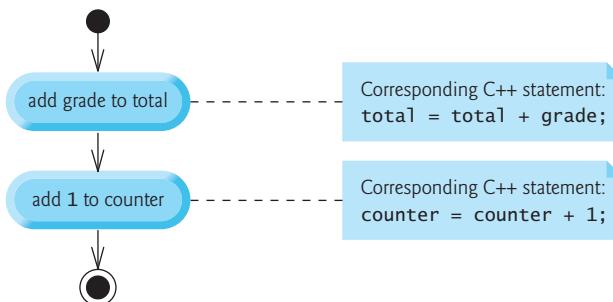
1. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

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.

Böhm 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.

3.4.1 Sequence Structure

The sequence structure is built into C++. Unless directed otherwise, statements execute one after the other in the order they appear in the program—that is, in sequence. The following UML² **activity diagram** illustrates a typical sequence structure in which two calculations are performed in order:



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**, representing the activity’s flow—that is, the order in which the actions should occur.

Like pseudocode, activity diagrams help you develop and represent algorithms. Activity diagrams clearly show how control structures operate.

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 program that implements the activities illustrated in this activity diagram first adds grade to total, then adds 1 to counter.

2. We use the UML (Unified Modeling Language) in this chapter and Chapter 4 to show the flow of control in control statements, then use UML again in Chapters 9–10 when we present custom class development.

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

3.4.2 Selection Statements

C++ has three types of **selection statements**. The **if** selection 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** selection 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** selection statement (Chapter 4) performs one of many different actions (or groups 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).

3.4.3 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 their action (or group of actions) 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 its action (or group of actions) 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** is a C++ keyword. Keywords cannot be used as identifiers, such as variable names, and contain only lowercase letters (and sometimes underscores). The following table shows the complete list of C++ keywords:

C++ keywords				
alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	const_cast	constexpr

C++ keywords (Cont.)

continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	final	float
for	friend	goto	if	import
inline	int	long	module	mutable
namespace	new	noexcept	not	not_eq
nullptr	operator	or	or_eq	override
private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned
using	void	volatile	virtual	wchar_t
while	xor	xor_eq		

Keywords new in C++20

char8_t	concept	constexpr	constinit	co_await
co_return	co_yield	requires		

3.4.4 Summary of Control Statements

C++ has only three kinds of control structures, which from this point forward, we refer to as control statements:

- sequence,
- selection (`if`, `if...else` and `switch`) and
- iteration (`while`, `do...while`, `for` and range-based `for`).

You form every program by combining these statements as appropriate for the algorithm you're implementing. We can model each control statement as an activity diagram. Each diagram contains initial and final states representing a control statement's entry and exit points. **Single-entry/single-exit control statements** make it easy to build readable 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.



Checkpoint

I (Fill-in) Böhm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—_____ , _____ and _____ structure.

Answer: sequence, selection, iteration.

- 2 (True/False)** The `while`, `for` and `do...while` statements perform their action (or group of actions) zero or more times.

Answer: False. The `while` and `for` statements perform their action (or group of actions) zero or more times. The `do...while` statement performs its action (or group of actions) one or more times.

- 3 (Fill-in)** _____ control statements make it easy to build readable programs by connecting the exit point of one to the entry point of the next using control-statement stacking.
Answer: Single-entry/single-exit.

3.5 if Single-Selection Statement

We introduced the `if` single-selection statement briefly in Section 2.7. Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The following pseudocode represents an `if` statement that tests the **condition** “student’s grade is greater than or equal to 60”:

```
if student's grade is greater than or equal to 60
    print "Passed"
```

If this condition is true, the statement prints “Passed”—otherwise, the `print` statement is ignored. Indenting the second line of this selection statement is optional but recommended for clarity—it emphasizes that the `print` statement is in the `if` statement’s body.

The preceding pseudocode may be written in C++ as

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

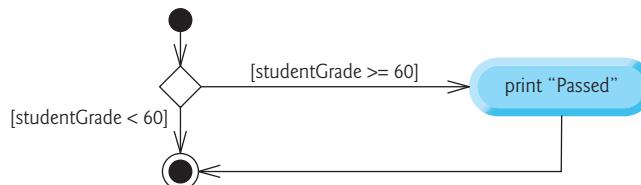
The C++ code corresponds closely to the pseudocode.

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 is a C++ keyword. The compiler can implicitly convert `true` to 1 and `false` to 0.

UML Activity Diagram for an if Statement

The following UML activity 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 UML 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.



Checkpoint

- 1 (*Pseudocode*) Write a pseudocode **if statement** that determines whether a count is greater than 10 and, if so, prints “Count is greater than 10.”

Answer:

```
if count grade is greater than 10
    print "Count is greater than 10."
```

- 2 (*Code*) Write an **if** statement that implements the preceding exercise’s pseudocode. Assume the **int** variable **count** already exists.

Answer:

```
if (count > 10) {
    cout << "Count is greater than 10.";
}
```

- 3 (*Fill-in*) A diamond, or decision symbol, in an activity diagram indicates that a decision is to be made. The workflow continues along a path determined by the symbol’s associated _____.

Answer: guard conditions.

3.6 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 following pseudocode represents an **if...else** statement that prints “Passed” if the student’s grade is greater than or equal to 60 but prints “Failed” if it’s less than 60:

```
if student's grade is greater than or equal to 60
    print "Passed"
else
    print "Failed"
```

In either case, after printing occurs, the next statement in sequence is “performed.” The preceding **if...else pseudocode statement** can be written in C++ as

```

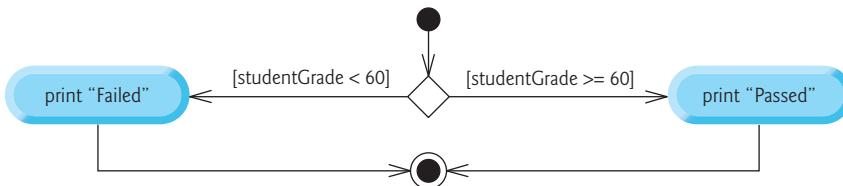
if (studentGrade >= 60) {
    cout << "Passed";
}
else {
    cout << "Failed";
}

```

The `if` and `else` bodies are both indented equally. Whatever indentation convention you choose should be applied consistently throughout your programs. If there are several indentation levels, each should be indented the same additional amount of space. We prefer three-space indents. IDEs often auto-indent the code for you.

UML Activity Diagram for an if...else Statement

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



3.6.1 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 pseudocode represents a nested `if...else` statement that 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 student's grade is greater than or equal to 90
    print "A"
else
    if student's grade is greater than or equal to 80
        print "B"
    else
        if student's grade is greater than or equal to 70
            print "C"
        else
            if student's grade is greater than or equal to 60
                print "D"
            else
                print "F"

```

This pseudocode may be written in C++ as:

```
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. In a nested `if...else` statement, ensure that you test for all possible cases.

The preceding nested `if...else` statement also can be written in the following form, which is identical but uses fewer braces, less spacing and indentation:

```
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 sometimes can force lines to wrap. Throughout the text, we enclose control-statement bodies in braces (`{` and `}`), which avoids a logic error called the “dangling-`else`” problem. We investigate this  Err problem in Exercises 3.13–3.15.

3.6.2 Blocks

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

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

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

If `studentGrade` is less than 60, the program executes both statements in the body of the `else` and prints

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

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

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

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

Syntax and Logic Errors

The compiler catches **syntax errors**, such as when one brace in a block is left out of the program. A **logic error**, such as an incorrect calculation, has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.

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 simply a semicolon (`;`) where a statement typically would be. An empty statement has no effect.



Placing a semicolon after the parenthesized condition in an `if` or `if...else` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if`-part contains a body statement).

3.6.3 Conditional Operator (`? :`)

C++ provides the **conditional operator** (`? :`), which 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 following statement prints the conditional expression's value:

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

The operand to the left of the ? is a condition. The second operand (between the ? and :) is the conditional expression's value if the condition is true. The operand to the right of the : is the conditional expression's value 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 3.6. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses.



Checkpoint

- 1 (*Pseudocode*) Write a pseudocode if...else statement that determines whether a counter is greater than 10. If so, the statement should print "counter is greater than 10." Otherwise, it should print "counter is less than or equal to 10."

Answer:

```
if counter is greater than 10
    print "counter is greater than 10."
else
    print "counter is less than or equal to 10."
```

- 2 (*Code*) Write an if...else statement that implements the preceding exercise's pseudocode. Assume the int variable counter already exists.

Answer:

```
if (counter > 10) {
    cout << "counter is greater than 10.";
}
else {
    cout << "counter is less than or equal to 10.";
}
```

- 3 (*Code*) Write a nested if...else statement that displays whether the value of counter is greater than 10, equal to 10 or less than 10.

Answer:

```
if (counter > 10) {
    cout << "counter is greater than 10.";
}
else if (counter == 10) {
    cout << "counter is equal to 10.";
}
else {
    cout << "counter is less than 10.";
}
```

3.7 while Iteration Statement

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

```
while there are more items on my shopping list
    purchase next item and cross it off my list
```

describes the iteration during a shopping trip. The condition “there are more items on my shopping list” may be *true* or *false*. If it’s *true*, the action “purchase next item and cross it off my list” is performed. This action will be performed *repeatedly* while the condition remains *true*. The statement(s) contained in the **while iteration statement** constitute its body, which may be a single statement or a block. Eventually, the condition will become *false* (when the shopping list’s last item has been purchased and crossed off). At this point, the iteration terminates, and the first statement after the iteration statement executes.

As a **while iteration statement** example, consider the following program segment that finds the first power of 3 larger than 100:

```
int product{3};

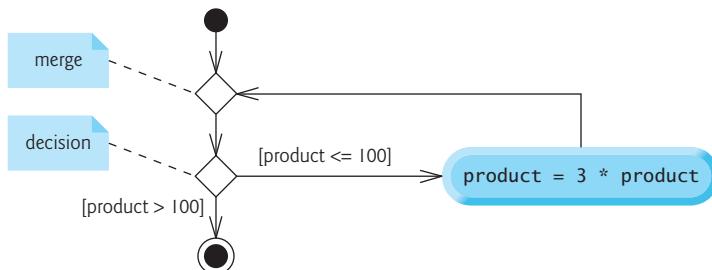
while (product <= 100) {
    product = 3 * product;
}
```



After this **while** statement executes, the variable **product** contains the result. 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. Not providing in a **while** statement’s body an action that eventually causes the condition to become false results in a logic error called an **infinite loop** (the loop never terminates).

UML Activity Diagram for a while Statement

The following **while** statement UML activity diagram introduces the **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 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 decision. Also, each arrow pointing out of a decision symbol has a guard condition. A merge symbol has two or more transition arrows pointing to it, and only one pointing from it to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.



Checkpoint

- 1 *(True/False)* A UML merge symbol has one transition arrow pointing to the diamond and two or more pointing out from it.

Answer: False. A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it (each with a guard condition) to indicate possible transitions. A merge symbol has two or more transition arrows pointing to it, and only one pointing from it to indicate multiple activity flows merging to continue the activity.

- 2 *(Code)* Using this section's while statement as a guide, write code that determines the first power of 2 greater than one million. Add a counter that determines what the first n is for $2^n > 1,000,000$.

Answer:

```
int product{2};  
int n{1};  
  
while (product <= 1000000) {  
    product = 2 * product;  
    n = n + 1;  
}  
  
cout << "First power of 2 greater than 1000000 is " << product  
    << "\nwhich is 2 to the power of " << n << "\n";
```

First power of 2 greater than 1000000 is 1048576
which is 2 to the power of 20

3.8 Formulating Algorithms: Counter-Controlled Iteration

To illustrate how algorithms are developed, we solve two variations of a problem that averages student grades. Consider the following problem statement:

A class of ten students took a quiz. The quiz grades (integers in the range 0–100) 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, total all the grades entered, perform the averaging calculation and print the result.

3.8.1 Pseudocode Algorithm with Counter-Controlled Iteration

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use **counter-controlled iteration** to input the grades one at a time. This technique uses a variable called a **counter** (or **control variable**) to control the number of times a set of statements will execute. Counter-controlled iteration is often called **definite iteration** because the number of iterations is known *before* the loop begins executing. In this example, iteration terminates when the counter exceeds 10. This section presents the following fully developed pseudocode algorithm and a corresponding C++ program (Fig. 3.1) that implements the algorithm. In Section 3.9, we demonstrate how to develop pseudocode algorithms from scratch.

```

1 set total to zero
2 set grade counter to one
3
4 while grade counter is less than or equal to ten
5   prompt the user to enter the next grade
6   input the next grade
7   add the grade into the total
8   add one to the grade counter
9
10 set the class average to the total divided by ten
11 print the class average

```

Note the references in the preceding algorithm to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A counter is a variable used to count—the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals normally are initialized to zero before being used in a program. In pseudocode, we do *not* use braces around the statements that form the pseudocode **while**'s body, but you could.



Experience has shown that the most challenging part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working C++ program from it is usually straightforward.

3.8.2 Implementing Counter-Controlled Iteration

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

```

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

```

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

```
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 << "\n";
26 }
```

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

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

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

Local Variables in main

Lines 8, 9, 14 and 21 declare `int` variables `total`, `gradeCounter`, `grade` and `average`, respectively. Variable `grade` stores the user input. A variable declared in a block (such as a function's body) is a **local variable** that can be used only from the line of its declaration to the closing right brace of the block. A local variable's declaration must appear before the variable is used. Variable `grade`—declared at line 14 the `while` loop's body—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. Initialize each total and counter, either in its declaration or in an assignment statement. Totals are typically initialized to 0 and counters are typically initialized to 0 or 1, depending on their use. We'll show examples of when to use 0 or 1.

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 vari-

able `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.

3.8.3 Integer Division and Truncation

This example's average calculation produces an `int` result. The program's sample execution shows that the sum of the grades is 846—when divided by 10, this should yield 84.6. Numbers like 84.6 containing decimal points are **floating-point numbers**. However, in the class-average program, `total / 10` produces 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. The next section shows how to obtain a floating-point result from the averaging calculation. For example, `7 / 4` yields 1.75 in conventional arithmetic but truncates to 1 in integer arithmetic rather than rounding to 2. Assuming that integer division rounds (rather than truncates) can lead to incorrect results.



3.8.4 Arithmetic Overflow

In Fig. 3.1, line 16

```
total = total + grade; // add grade to total
```

adds each `grade` entered by the user to the `total`. Even this simple statement has a potential problem—adding the integers could result in a value too large to store in an `int` variable. This is known as **arithmetic overflow** and causes **undefined behavior**, which can lead to unintended results and security problems. See, for example,



https://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications

The following expressions return your system's maximum and minimum `int` values:

```
std::numeric_limits<int>::max()
std::numeric_limits<int>::min()
```

Each uses class `numeric_limits` from the C++ standard library header `<limits>`. You may use similar expressions for any integral or floating-point type by replacing `int` in these expressions.

3.8.5 Input Validation

When a program receives input from the user, various problems might occur. For example, line 15 of Fig. 3.1

```
cin >> grade; // input next grade
```



assumes that the user will enter an integer grade from 0 to 100 but the user could enter:

- an integer less than 0,
- an integer greater than 100,
- an integer outside the range of values that can be stored in an `int` variable,
- a number containing a decimal point or
- a value containing letters or special symbols that do not even represent a number.



Industrial-strength programs must test for all possible erroneous cases to ensure valid inputs. A program that inputs grades should validate them using **range checking** to ensure they are values from 0 to 100. You can then ask the user to re-enter any out-of-range value. If a program requires inputs from a specific set of values (such as, non-sequential product codes), you should ensure that each input matches a value in the set.



Checkpoint

- 1 *(True/False)* Dividing two integers results in integer division—any fractional part of the calculation is rounded to the nearest integer.

Answer: False. Actually, any fractional part of the calculation is truncated.

- 2 *(Fill-in)* A variable declared in a(n) _____ (such as a function's body) is a local variable that can be used only from the line of its declaration to the corresponding closing right brace.

Answer: block.

- 3 *(Code)* Rewrite the initialization in the following code so the loop iterates 5 times.

```
int count{1};  
  
while (count < 5) {  
    count = count + 1;  
    cout << "count is :" << count << "\n";  
}
```

Answer:

```
int count{0};  
  
while (count < 5) {  
    count = count + 1;  
    cout << "count is :" << count << "\n";  
}
```

```
count is: 1  
count is: 2  
count is: 3  
count is: 4  
count is: 5
```

3.9 Formulating Algorithms: Sentinel-Controlled Iteration

Let's generalize Section 3.8'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 is 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. For this problem, we do not know 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 **sentinel value** (also called a **signal value** or a **flag value**) to indicate the end of data entry. The user enters grades until all legitimate grades have been entered. The user then enters 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 non-negative 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, excluding -1 , because it's the sentinel value and should not enter into the averaging calculation.

The user could enter -1 before entering grades, which means 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.

3.9.1 Top-Down, Stepwise Refinement: The Top and First Refinement

We approach this class-averaging program with a technique called **top-down, stepwise refinement**, which is essential to the development of well-structured programs. We begin with a pseudocode representation of the **top**—a single statement that conveys the overall function of the program:

determine the class average for the quiz

The top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys sufficient detail from which to write a C++ program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they'll be performed. This results in the following **first refinement**:

initialize variables
input, sum and count the quiz grades
calculate and print the class average



This refinement uses only statements in sequence—the steps listed should execute in order, one after the other. Like the **top**, each refinement is a complete specification of the algorithm—only the level of detail varies.

3.9.2 Proceeding to the Second Refinement

Sometimes, the first refinement in the top-down process might be all you need. Here, we must provide more detail, so we proceed to the **second refinement**, in which we commit to specific variables. We need

- a running total of the numbers,
- a count of how many numbers we've processed,
- a variable to receive each grade entered by the user and
- a variable to hold the calculated average.

The pseudocode statement

initialize variables

can be refined as follows:

initialize total to zero

initialize counter to zero

Only the variables *total* and *counter* need to be initialized before they're used. The variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized, as their values will be replaced by a calculation or user inputs. Many programmers, nevertheless, choose to initialize all variables—as you'll see later in the book, this can help avoid subtle errors.

The pseudocode statement

input, sum and count the quiz grades

requires iteration to input each grade. We do not know the number of grades in advance, so we'll use sentinel-controlled iteration. The user enters grades one at a time. After the last grade, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is:

prompt the user to enter the first grade

input the first grade (possibly the sentinel)

while the user has not yet entered the sentinel

add this grade into the running total

add one to the grade counter

prompt the user to enter the next grade

input the next grade (possibly the sentinel)

We simply indent the statements under the **while** to show that they belong to its body. Again, pseudocode is only an informal program development aid.

The pseudocode statement

calculate and print the class average

can be refined as follows:

if the counter is not equal to zero

set the average to the total divided by the counter

print the average

else

print "No grades were entered"

We're careful here to test for the possibility of **division by zero**—a **logic error** that, if undetected, would cause the program to fail or produce invalid output. 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. Here is the complete second refinement of the pseudocode for the class-average problem:



```

1 initialize total to zero
2 initialize counter to zero
3
4 prompt the user to enter the first grade
5 input the first grade (possibly the sentinel)
6
7 while the user has not yet entered the sentinel
8     add this grade into the running total
9     add one to the grade counter
10    prompt the user to enter the next grade
11    input the next grade (possibly the sentinel)
12
13 if the counter is not equal to zero
14     set the average to the total divided by the counter
15     print the average
16 else
17     print "No grades were entered"

```

We use blank lines to make the pseudocode more readable and indentation to emphasize the actions that should be placed in the control statements' bodies. The blank lines separate the algorithms into their phases and set off control statements; the indentation emphasizes the bodies of the control statements.

Our pseudocode algorithm solves the more general class-average problem. This algorithm required only two refinements—sometimes, more are needed. You terminate the top-down, stepwise refinement process when you feel your pseudocode algorithm has sufficient detail for you to convert the algorithm to C++.

Some programmers do not use program development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the final product. This may work for simple and familiar problems but can lead to significant errors and delays in large, complex projects.

3.9.3 Implementing Sentinel-Controlled Iteration

Figure 3.2 implements sentinel-controlled iteration. Although each grade entered by the user is an integer, the average calculation will likely produce a floating-point number, which an `int` cannot represent. C++ provides types `float`, `double` and `long double` to store floating-point numbers:

- A `double` variable typically stores numbers with larger magnitude and finer detail than a `float`—that is, more digits to the right of the decimal point, also known as the number's **precision**.
- Similarly, a `long double` variable typically stores values with larger magnitude and more precision than `double`.

To calculate a floating-point average in this example, we'll store the total as a `double`—C++'s default type for floating-point literals, such as `-7.2`, `0.0` or `98.6`.

```
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     double total{0.0}; // 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) { // avoid division by zero
31        // calculate average of grades
32        double average{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 << "\n";
39    }
40    else { // no grades were entered, so output appropriate message
41        cout << "No grades were entered\n";
42    }
43}
```

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

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

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

```
Enter grade or -1 to quit: -1
No grades were entered
```

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

Recall that integer division produces an integer result. This program declares the variable `total` as a `double` (line 9). As you'll see, this will cause the average calculation to produce a floating-point result. This program also stacks control statements in sequence—the `while` statement is followed in sequence by an `if...else` statement. Much of this program's code is identical to 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 the `double` variable `average`, which stores the class average as a floating-point number.

Compare this program's logic for sentinel-controlled iteration with Fig. 3.1's counter-controlled iteration:

- In counter-controlled iteration, each loop iteration (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 executes, adding the `grade` value to the `total` and incrementing the `gradeCounter`. Then lines 24–25 prompt for and input the next grade. At this point, program control reaches the loop's closing right brace (line 26). Execution continues by testing the `while`'s condition (line 19), using the most recently entered `grade` 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 processing it (i.e., adding it to the `total`). If `grade` contains the sentinel value, the loop terminates without adding `-1` to the `total`. In a sentinel-controlled loop, prompts should remind the user of the sentinel.

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.

3.9.4 Mixed-Type Expressions and Implicit Type Promotions

If at least one grade was entered, line 32

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

divides `total` by `gradeCounter` to calculate the average. Recall that `total` is a `double` and `gradeCounter` is an `int`. For arithmetic, the compiler knows how to evaluate only expres-

sions 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 `double` and `int`, C++ **promotes** any `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.

3.9.5 Formatting Floating-Point Numbers

Figure 3.2's formatting features are introduced here briefly. Chapter 19 explains them in depth. In Chapter 4, we'll introduce C++20's new text-formatting features that can express the formatting we show here more concisely.

Parameterized Stream Manipulator `setprecision`

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>` (line 4).

Nonparameterized Stream Manipulator `fixed`

The **non-parameterized stream manipulator `fixed`** (line 37) does not require an argument and indicates that floating-point values should be output in **fixed-point format**. This is as opposed to **scientific notation**³, which displays a number from 1.0 up to, but not including, 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 (that is, values with negative exponents).

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.

Stream manipulators `setprecision` and `fixed` perform **sticky settings**—all floating-point values formatted in your program will use those settings until you change them. Chapter 19 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 formatted value is **rounded** to the number of decimal positions specified by the current precision. The value in memory remains unaltered. For the precision 2, 87.946 and 67.543 would be rounded to 87.95 and 67.54, respectively.⁴

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

-
3. Formatting using scientific notation is discussed further in Chapter 19.
 4. In Fig. 3.2, if you do not specify `setprecision` and `fixed`, C++ uses four digits of precision by default. If you specify only `fixed`, C++ uses six digits of precision.



Checkpoint

1 (*True/False*) Any value may be used as a sentinel value.

Answer: False. You must choose a sentinel value that cannot be confused with an acceptable input value.

2 (*Fill-in*) For arithmetic on fundamental types, 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 _____ (also called _____) on selected operands.

Answer: promotion, implicit conversion.

3 (*Code*) Write a program segment that reads an arbitrary number of integer Fahrenheit temperatures from the user and calculates their average. Assume the user enters at least one temperature and temperatures are in the range –212 to 212. Display the average.

Answer:

```
double total{0.0};
int counter{0};

cout << "Enter a temperature or 9999 to quit: ";
int temperature;
cin >> temperature;

while (temperature != 9999) {
    total = total + temperature;
    counter = counter + 1;

    cout << "Enter a temperature or 9999 to quit: ";
    cin >> temperature;
}

double average{total / counter};
cout << "Average temperature is: " << average << "\n";
```

```
Enter a temperature or 9999 to quit: 12
Enter a temperature or 9999 to quit: -22
Enter a temperature or 9999 to quit: 17
Enter a temperature or 9999 to quit: 9999
Average temperature is: 2.33333
```

3.10 Formulating Algorithms: 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 **nesting** one control statement within another—the other structured way control statements can be connected.

3.10.1 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 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 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 "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 pass, print "Bonus to instructor!"

Problem Statement Observations

After reading the problem statement carefully, we make the following observations:

1. The program must process test results for 10 students. We can counter-controlled iteration because we know the number of test results in advance.
2. Each time the program reads a test result, it must determine whether it's a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume it's a 2. (Exercise 3.10 considers the consequences of this assumption.)
3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number who failed.
4. After the program has processed all the results, it must decide whether more than eight students passed the exam.

3.10.2 Top-Down, Stepwise Refinement: Pseudocode Representation of the Top

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

analyze exam results and decide whether a bonus should be paid

Once again, the top is a complete representation of the program, but we'll likely need several refinements before the pseudocode can evolve naturally into a C++ program.

3.10.3 Top-Down, Stepwise Refinement: First Refinement

Our first refinement is

initialize variables
input the 10 exam results, and count passes and failures
print a summary of the exam results and decide whether a bonus should be paid

Here, too, though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process, and a variable to store the user input. We do not need to initialize the variable that stores the user input because its value is read from the user during each loop iteration.

3.10.4 Top-Down, Stepwise Refinement: Second Refinement

The pseudocode statement

initialize variables

can be refined as follows:

initialize passes to zero
initialize failures to zero
initialize student counter to one

Notice that only the counters are initialized at the start of the algorithm.

The pseudocode statement

input the 10 exam results, and count passes and failures

requires a loop that successively inputs the result of each exam. We know in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., nested within the loop), a double-selection structure will determine whether each exam result is a pass or a failure, then increment the appropriate counter. The preceding pseudocode statement can be refined as follows:

```
while student counter is less than or equal to 10
    prompt the user to enter the next exam result
    input the next exam result
    if the student passed
        add one to passes
    else
        add one to failures
    add one to student counter
```

We use blank lines to isolate the **nested if...else statement**, which improves readability.

The pseudocode statement

print a summary of the exam results and decide whether a bonus should be paid

can be refined as follows:

```
print the number of passes
print the number of failures
if more than eight students passed
    print "Bonus to instructor!"
```

3.10.5 Complete Second Refinement of the Pseudocode

Here we show the complete second refinement. This pseudocode is now sufficiently refined for conversion to C++.

-
- 1 initialize passes to zero
 - 2 initialize failures to zero
 - 3 initialize student counter to one
 - 4

(continued on next page)

```
5  while student counter is less than or equal to 10
6      prompt the user to enter the next exam result
7      input the next exam result
8
9      if the student passed
10         add one to passes
11     else
12         add one to failures
13
14     add one to student counter
15
16    print the number of passes
17    print the number of failures
18
19    if more than eight students passed
20        print "Bonus to instructor!"
```

3.10.6 Implementing the Program

Figure 3.3 implements the program with counter-controlled iteration and shows two sample executions. Lines 8–10 and 16 declare the variables we use to process the examination results. Lines 13–29 loop 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 20–25) for processing each result is nested in the `while` statement. If the `result` is 1, the `if...else` statement increments `passes`; otherwise, it assumes the `result` is 2 and increments `failures`.⁵ Line 28 increments `studentCounter` before the loop condition is tested again at line 13.

```
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}
```

Fig. 3.3 | Analysis of examination results using nested control statements. (Part I of 2.)

5. Assuming `result` is 2 could be a bad assumption if the user enters invalid data. We'll discuss data-validation techniques later.

```

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 << "\n";
33
34 // determine whether more than 8 students passed
35 if (passes > 8) {
36     cout << "Bonus to instructor!\n";
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
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```

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

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

After 10 values have been input, the loop terminates, and line 32 displays the number of passes and failures. The `if` statement in lines 35–37 determines whether more than eight students passed the exam and, if so, outputs the message "Bonus to instructor!"

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 give the instructor a bonus.

3.10.7 Preventing Narrowing Conversions with Braced Initialization

Consider the braced initialization in line 10 of Fig. 3.3:

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

which also can be written as

```
int studentCounter = 1;
```

For fundamental-type variables, braced initializers prevent **narrowing conversions** that could result in **data loss**. For example, the following declaration attempts to assign the `double` value 12.6 to the `int` variable `x`:

```
int x = 12.6;
```

Here, C++ converts the `double` value to an `int` by truncating the floating-point part (.6), so this declaration assigns 12 to `x`. The narrowing conversion loses data. Compilers typically issue a warning for this but still compile the code.

However, using braced initialization, as in

```
int x{12.6};
```

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

The C++ standard document does not specify error-message wording. For the preceding declaration, Visual Studio gives the error

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

GNU C++ gives the error

```
error: narrowing conversion of '1.26e+1' from  
'double' to 'int' [-Wnarrowing]
```

and the clang++ compiler gives the error

```
error: type 'double' cannot be narrowed to 'int' in initializer list  
[-Wc++11-narrowing]
```

followed by more explanation

```
warning: implicit conversion from 'double' to 'int' changes value  
from 12.6 to 12 [-Wliteral-conversion]
```

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

A Look Back at Fig. 3.1

You might think that the following statement from Fig. 3.1

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

contains a narrowing conversion, but total and 10 are both `int` values, so the initializer value is an `int`. If total were a `double` in the preceding statement or we used the `double` literal 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.

**Checkpoint**

- 1** *(Code)* Write an initialization statement for the `int` variable `temperature`. Attempt to initialize the variable with the `double` value 98.6, but write the statement in a manner that will prevent a narrowing conversion.

Answer: `int temperature{98.6};`

- 2** *(Code)* What is the initial value of `average` in the following statement?

```
int average{128 / 3};
```

Answer: In this statement, 128 and 3 are both integers, so `128 / 3` produces an integer result. The variable `average`'s value will be 42.

- 3** *(Code)* Write a program segment that loops three times. In each iteration, input an integer from the user, then use a nested `if...else` statement to determine and display whether the number is even or odd.

Answer:

```
int counter{1};

while (counter <= 3) {
    cout << "Enter an integer: ";
    int number;
    cin >> number;

    if (number % 2 == 0) {
        cout << number << " is even\n";
    }
    else {
        cout << number << " is odd\n";
    }

    counter = counter + 1;
}
```

```
Enter an integer: 2
2 is even
Enter an integer: 3
3 is odd
Enter an integer: 5
5 is odd
```

3.11 Compound Assignment Operators

You can abbreviate the statement

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

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

```
c += 3;
```

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

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



Checkpoint

- I (Code) Write a statement that uses a compound assignment operator to divide the variable `total` by 5.

Answer: `total /= 5;`

3.12 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 placed before (prefixed to) a variable is called the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator placed after (postfixed to) a variable is called 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 value before the increment (or decrement) is used in the expression. Unlike binary operators, the unary increment and decrement operators by convention should be placed next to their operands, with no intervening spaces.

Prefix Increment vs. Postfix Increment

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

```

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

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

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

Fig. 3.4 | Prefix increment and postfix increment operators.

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

original value (5) is output, then c's value is incremented (to 6). Thus, line 10 outputs c's initial value (5) again. Line 11 outputs c's new value (6) to prove that the variable's value was 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 c is still 6 after line 18 executes.

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

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

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

can be written more concisely with compound assignment operators as

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

with prefix increment operators as

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

or with postfix increment operators as

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

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

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a compilation 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 *right-to-left*. All other operators in this table group *left-to-right*.

Operators	Grouping
<code>++ (postfix) -- (postfix)</code>	left to right
<code>+ - ++ (prefix) -- (prefix)</code>	right to left
<code>* / %</code>	left to right
<code>+ -</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>?:</code>	right to left
<code>= += -= *= /= %=</code>	right to left



Checkpoint

- 1 *(Code)* Assume the `int` variable `counter` has the value 10. What are the values of `result` and `counter` after this statement executes?

```
int result{counter--};
```

Answer: The initialization expression uses the postdecrement operator, so `result` will contain 10 and `counter` will contain 9.

- 2 *(Code)* Assume the `int` variable `counter` has the value 10. What are the values of `result` and `counter` after this statement executes?

```
int result{++counter};
```

Answer: The initialization expression uses the preincrement operator, so both `result` and `counter` will contain 11.

3.13 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.⁶ In the next section, we'll show one way to achieve portability.

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"

6. The integer types in the header `<cstdint>` (<https://en.cppreference.com/w/cpp/types/integer>) can be used to ensure that integer variables are correctly sized for your application across platforms.

requirements create portability challenges but allow compiler implementers to optimize performance by matching fundamental types sizes to your machine's hardware.



Checkpoint

- 1 (*True/False*) All computers represent an `int` in the same number of bytes.

Answer: False. 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).

- 2 (*True/False*) The C++ standard 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.

Answer: True.

3.14 Objects Natural Case Study: Super-Sized Integers

The explosive growth of Internet communications and data storage on Internet-connected devices has dramatically increased the importance of privacy and security. Cryptography has been used for thousands of years to encode data, making it difficult (and hopefully impossible) for unauthorized users to read,^{7,8} which is critically important in today's connected world. Most websites (including `deitel.com`) now use HTTPS (HyperText Transfer Protocol Secure) to encrypt and decrypt your Internet interactions.



Some Applications Need Integers Outside a `long long` Integer's Range

Many cryptography algorithms perform calculations with integers far larger than we can store using even C++'s `long long` type. On most systems, a `long long` integer can store values in the range $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$ —a maximum of 19 decimal digits (which can represent integers in the quintillions).

One of the most popular encryption/decryption schemes is **RSA Public-Key Cryptography**, commonly used to secure data transmitted over the Internet.^{9,10,11} Industrial-strength RSA implementations work with **enormous prime numbers consisting of hundreds of digits**.



A key reason why RSA is so secure is the sheer amount of time required to factor the product of those primes, even for today's most powerful supercomputers. Public-key cryptography also is used to secure the blockchain technology behind cryptocurrencies like Bitcoin.¹² Later in the book, we'll discuss and implement public-key cryptography with RSA.



7. "Cryptography." January 15, 2023. <https://en.wikipedia.org/wiki/Cryptography#History>.
8. Binance Academy, "History of Cryptography." Accessed April 14, 2023. <https://www.binance.vision/security/history-of-cryptography>.
9. "RSA (Cryptosystem)." Accessed April 14, 2023. [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)).
10. "RSA Algorithm." Accessed April 14, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.
11. K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," November 2016. Accessed April 14, 2023. <https://tools.ietf.org/html/rfc8017>.
12. Sarah Rothrie, "How Blockchain Cryptography Is Fighting the Rise of Quantum Machines," December 28, 2018. Accessed April 14, 2023. <https://coincentral.com/blockchain-cryptography-quantum-machines/>.

Integers Outside the `long long` Range Require Custom Programming

So how can we manipulate huge integers with arbitrary numbers of digits? The C++ standard library does not provide such capabilities, so applications requiring integers outside `long long`'s range require custom programming. We could use the C++' class mechanism (mentioned in Section 1.9 and covered in detail in Chapter 9) to develop a custom class that handles huge-integer arithmetic—but that could take months of effort. The Objects Natural approach encourages us to check first for free, open-source class libraries to see if such a “huge-integer” class already exists.

Huge Integers with the Boost Multiprecision Open Source Library's `cpp_int` Class

Indeed, there are many preexisting C++ open-source classes for creating and manipulating huge integers. For this example, we'll demonstrate the [Boost Multiprecision open source library](#)'s `cpp_int` class from:

```
https://github.com/boostorg/multiprecision/
```

For your convenience, we placed this library in the `examples` folder's `libraries` subfolder.¹³ Boost provides 168 open-source C++ libraries.¹⁴ It also serves as a “breeding ground” for new capabilities that often are incorporated into the C++ standard libraries.¹⁵

With the Objects Natural approach in Fig. 3.5, we'll conveniently use class `cpp_int` without having to understand how it's implemented.^{16,17} We include its header (line 4), easily create `cpp_int` objects (lines 11–12), then conveniently manipulate them in our code with familiar arithmetic operations (lines 22–29). After we present the code, we show how to compile and run this example on each of our preferred compilers.

-
13. At the time of this writing, the latest version of this library was 1.80. The Boost libraries use the open-source Boost Software License 1.0 terms, which allow distribution and commercial and private use. See the license's full terms in the library's `LICENSE` file:
<https://github.com/boostorg/multiprecision/blob/develop/LICENSE>.
 14. “Boost 1.80.0 Library Documentation.” Accessed August 31, 2022. https://www.boost.org/doc/libs/1_80_0/.
 15. Kennytm, Answer to “Which Boost Features Overlap with C++11?” Accessed August 31, 2022. <https://stackoverflow.com/a/8852421>. This StackOverflow answer lists Modern C++ libraries and language features that evolved from the Boost libraries.
 16. If you drive a car, you use it effectively without having to understand how car engines, steering, brakes and transmission systems work internally. After you get deeper into C++, you might want to study `cpp_int`'s source code (approximately 1,000 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 huge-integer class.
 17. You might say, “I insist on knowing how everything I use in C++ is implemented.” Reflect for a moment on the fact that you're using a C++ compiler to translate your C++ programs into your computer's machine language. Do you know how the compiler is implemented? Are you sure that it correctly translates each of your C++ programs so they'll run properly and produce correct results?

Fig. 3.5 | Conveniently creating and manipulating super-sized integers with objects of the Boost Multiprecision library's `cpp_int` class.

using Declaration

Section 2.7 showed that a `using` declaration can eliminate the need to repeat the "std::" prefix for C++ standard library components, such as `std::cin` and `std::cout`. The Boost Multiprecision library's capabilities must be similarly qualified with

```
boost::multiprecision::
```

as in

```
boost::multiprecision::cpp_int
```

Line 7's `using` declaration enables you to use the class name `cpp_int` rather than the fully qualified class name `boost::multiprecision::cpp_int`.

Initializing `cpp_ints`

Lines 11 and 12 initialize the `cpp_ints` `value1` and `value2` with a `string` and a `long long`, respectively. For cases in which the initializer value for a `cpp_int` exceeds the maximum integer values C++ supports, such as the 30-digit integer in line 11, you initialize a `cpp_int` with a `string` representing an integer value. You may also use any valid integer as an initializer, as in line 12, where we chose to use the maximum value of a `long long`.¹⁸ The `LL` ("el el") suffix at the end of the value indicates that it's a `long long` integer. Line 13 also defines the fundamental type `int` variable `value3`, which we'll use momentarily. Lines 15–18 display these variables' initial values.

Performing `cpp_int` Arithmetic as Simply as Arithmetic with Fundamental Integer Types

Class `cpp_int` supports the same arithmetic operations as C++'s `int`, `long` and `long long` fundamental types. Lines 21–24 demonstrate addition, subtraction and multiplication with `cpp_ints` `value1` and `value2`.¹⁹ **`cpp_int` correctly handles integers of any number of digits**. Each result is a value with more digits than can be represented by C++'s fundamental integer types. The multiplication in line 24, for example, produces a 48-digit integer. Though a `cpp_int` can represent any integer value, it is not built into your system's hardware (i.e., it's not a fundamental type). So calculations with `cpp_int` run (possibly much) slower than those with fundamental integer types.



How Do C++'s Arithmetic Operators and the `<<` Operator Understand `cpp_int`?

The compiler already knows how to use arithmetic operators and the `<<` operator with fundamental numeric types, but it has to be taught how to handle those operators for custom `cpp_int` class objects. Chapter 11 discusses in detail how to do this with a powerful technique called operator overloading.

-
- 18. 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 simply used the literal value 9223372036854775807 in Fig. 3.5.
 - 19. Exercise 3.27 asks you to try using `cpp_int` objects with the division (/) and modulus (%) operators, as well as with Boost Multiprecision arithmetic functions `pow` and `sqrt`.

Performing Mixed-Type Arithmetic Between a `cpp_int` and an Integer

Class `cpp_int` also supports mixed-type expressions containing `cpp_int`s and integers of C++'s fundamental types. Lines 27–29 demonstrate this by multiplying the `cpp_int` object `value1` by the `int` variable `value3`, and by the `int` value 17.

Compiling and Running the Example in Microsoft Visual Studio

In Microsoft Visual Studio:

1. Create a new project, as described in Section 1.11.
2. In the **Solution Explorer**, right-click the project's **Source Files** folder and select **Add > Existing Item....**
3. Navigate to the `ch03` folder, select `fig03_05.cpp` and click **Add**.
4. In the **Solution Explorer**, right-click the project's name and select **Properties....**
5. Under **Configuration Properties**, expand **C/C++**, select **General**, select **Additional Include Directories**, then click the down arrow and select **Edit...** to display the **Additional Include Directories** dialog.
6. Click the yellow folder icon to add a new entry, then click the **...** button to open the **Select Directory** dialog.
7. Navigate into the `libraries\multiprecision-Boost_1_80_0` folder on your system and select `include`, then click **Select Folder**.
8. Click **OK** to close the **Additional Include Directories** dialog, then click **OK** to close your project's properties dialog.
9. Type `Ctrl + F5` to compile and run the program.²⁰

Compiling and Running the Example in GNU g++ or clang++

For GNU g++:

1. At your command line, change to the `ch03` folder.
2. To compile the program, type the following command (on one line)—the `-I` option specifies additional folders in which the compiler should search for header files:
`g++ -std=c++20 -I ..\libraries\multiprecision-Boost_1_80_0\include fig03_05.cpp -o fig03_05`
3. Type the following command to execute the program:
`./fig03_05`

For clang++, perform the same steps, but replace `g++` with `clang++`. On some platforms, the compiler command may simply be `clang++-16` rather than `clang++`.

Compiling and Running the Example in Apple Xcode

In Apple Xcode:

1. Create a new project, as described in Section 1.11, and delete `main.cpp`.
2. Drag `fig03_05.cpp` from the `ch03` folder in the Finder onto your project's source-code folder in Xcode, then click **Finish** in the dialog that appears.

20. Visual C++ may issue warnings for code in class `cpp_int`.

3. Select your project name in the **Project Navigator**, then in the **Build Settings** tab under **Search Paths**, double-click to the right of **Header Search Paths**, then enter the full path to the `libraries/multiprecision-Boost_1_80_0/include` folder on your system. For our system, this was the following location—you should replace *account* with your account’s username:

/Users/account/Documents/examples/libraries/
multiprecision-Boost_1_80_0/include

4. Type `⌘ + R` to compile and run the program.



Checkpoint

- | (True/False) Calculations with `cpp_int` objects perform about as well as with the built-in C++ integer types.

Answer: False. Though a `cpp_int` can represent any integer, it does not match your system's hardware. So you'll sacrifice performance in exchange for the flexibility `cpp_int` provides.

- 2 (Code)** Calculate and display the cube of the integer 100,000,000,000,000 (100 trillion) using the multiplication operator.

Answer:

```
cpp_int value{"10000000000000000"};
cout << "      value is: " << value <<
    << "\nvalue cubed is: " << value * value * value << "\n";
```

3.15 Wrap-Up

In this chapter, we developed algorithms using pseudocode and top-down, stepwise refinement. You saw that 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-controlled and sentinel-controlled iteration. We used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced the compound assignment operators and the increment and decrement operators.

We discussed why C++'s fundamental types are not guaranteed to be portable across platforms. Then with our Objects Natural approach, we conveniently created and manipulated objects of the open-source Boost Multiprecision library's `cpp_int` class to perform integer arithmetic with values outside the range supported by our systems. Chapter 4 continues our control-statements discussion, introducing the `for`, `do...while` and `switch` statements. It also introduces the logical operators for creating compound conditions.

Exercises

- 3.1** (*Correct the Code Errors*) Identify and correct the error(s) in each of the following program segments:

```

a) if (age >= 65) {
    cout << "Age is greater than or equal to 65\n";
}
else {
    cout << "Age is less than 65\n";
}
b) int x{1};
int total;

while (x <= 10)
    total += x;
    ++x;
c) int y{10};

while (y > 0) {
    cout << y << "\n";
    ++y;
}

```

3.2 (*What Does This Program Do?*) What does the following program print?

```

1 // ex03_02.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     int x{1};
7     int total{0};
8
9     while (x <= 10) {
10         int y = x * x;
11         cout << y << "\n";
12         total += y;
13         ++x;
14     }
15
16     cout << "Total is " << total << "\n";
17 }
```

For Exercises 3.3–3.6, perform each of these steps:

- Read the problem statement.
- Formulate the algorithm using pseudocode and top-down, stepwise refinement.
- Write a C++ program.
- Test, debug and execute the C++ program.

3.3 (*Gas Mileage*) Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several trips by recording miles driven and gallons used for each trip. Develop a C++ program that uses a `while` statement to input the miles driven and gallons used for each trip. The program should calculate and display the miles per gal-

lon obtained for each trip and print the combined miles per gallon obtained for all tankfuls up to this point.

```
Enter miles driven (-1 to quit): 287
Enter gallons used: 13
MPG this trip: 22.076923
Total MPG: 22.076923

Enter miles driven (-1 to quit): 200
Enter gallons used: 10
MPG this trip: 20.000000
Total MPG: 21.173913

Enter the miles driven (-1 to quit): 120
Enter gallons used: 5
MPG this trip: 24.000000
Total MPG: 21.678571

Enter the miles used (-1 to quit): -1
```

3.4 (Credit Limits) Develop a C++ program to determine whether a clothing-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

- Account number (an integer)
- Balance at the beginning of the month
- Total of all items charged by this customer this month
- Total of all credits applied to this customer's account this month
- Allowed credit limit

The program should use a `while` statement to input each of these facts, calculate the new balance (= beginning balance + charges – credits) and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit Limit Exceeded."

```
Enter account number (or -1 to quit): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
New balance is 5894.78
Account: 100
Credit limit: 5500.00
Balance: 5894.78
Credit Limit Exceeded.

Enter Account Number (or -1 to quit): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
New balance is 802.45

Enter Account Number (or -1 to quit): -1
```

3.5 (*Sales-Commission Calculator*) A large company pays its salespeople on a commission basis. The salespeople each receive \$200 per week plus 9% of their weekly gross sales. For example, a salesperson who sells \$5000 worth of chemicals receives \$200 plus 9% of \$5000 for a total of \$650. Develop a C++ program that uses a `while` statement to input each salesperson's gross sales for last week and calculates and displays that salesperson's earnings. Process one salesperson's figures at a time.

```
Enter sales in dollars (-1 to end): 5000
Salary is: $650.00
Enter sales in dollars (-1 to end): 6000
Salary is: $740.00
Enter sales in dollars (-1 to end): 7000
Salary is: $830.00
Enter sales in dollars (-1 to end): -1
```

3.6 (*Salary Calculator*) Develop a C++ program that uses a `while` statement to determine the gross pay for each of several employees. The company pays “straight time” for the first 40 hours worked by each employee and pays “time-and-a-half” for all hours worked over 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee’s gross pay.

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the employee: 10.00
Salary is $390.00
Enter hours worked (-1 to end): 40
Enter hourly rate of the employee: 10.00
Salary is $400.00
Enter hours worked (-1 to end): 41
Enter hourly rate of the employee: 10.00
Salary is $415.00
Enter hours worked (-1 to end): -1
```

3.7 (*Find the Largest*) The process of finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest inputs the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a C++ program that uses a `while` statement to determine and print the largest of 10 integers input by the user. Your program should use three variables, as follows:

- counter—Used to count to 10 to keep track of how many numbers have been input.
- number—The current number input to the program.
- largest—The largest number found so far.

3.8 (*Tabular Output*) Write a C++ program that uses a `while` statement and the tab escape sequence `\t` to print the following table of values:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

3.9 (Find the Two Largest Numbers) Using an approach similar to Exercise 3.7, find the two largest values among the 10 numbers. [Note: You must input each number only once.]

3.10 (Validating User Input) The examination results program of Fig. 3.3 assumes that any value input by the user that's not a 1 must be a 2. Modify the application to validate its inputs. If a value entered is not 1 or 2, keep looping until the user enters a correct value.

3.11 (What Does This Program Do?) What does the following program print?

```

1 // ex03_11.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     int count{1};
7
8     while (count <= 10) {
9         cout << (count % 2 == 1 ? "****" : "++++++") << "\n";
10        ++count;
11    }
12 }
```

3.12 (What Does This Program Do?) What does the following program print?

```

1 // ex03_12.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     int row{10};
7
8     while (row >= 1) {
9         int column{1};
10
11        while (column <= 10) {
12            cout << (row % 2 == 1 ? "<" : ">");
13            ++column;
14        }
15
16        --row;
17        cout << "\n";
18    }
19 }
```

3.13 (Dangling-else Problem) C++ compilers always associate an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`). This behavior can lead to what is referred to as the **dangling-else problem**. The indentation of the nested statement

```
if (x > 5)
    if (y > 5)
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

appears to indicate that if `x` is greater than 5, the nested `if` statement determines whether `y` is also greater than 5. If so, the statement outputs the string "`x and y are > 5`". Otherwise, it appears that if `x` is not greater than 5, the `else` part of the `if...else` outputs the string "`x is <= 5`". Beware! This nested `if...else` statement does *not* execute as it appears. The compiler actually interprets the statement as

```
if (x > 5)
    if (y > 5)
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

in which the body of the first `if` is a *nested if...else*. The outer `if` statement tests whether `x` is greater than 5. If so, execution continues by testing whether `y` is also greater than 5. If the second condition is *true*, the proper string—"`x and y are > 5`"—is displayed. However, if the second condition is *false*, the string "`x is <= 5`" is displayed, even though we know that `x` is greater than 5. Equally bad, if the outer `if` statement's condition is *false*, the inner `if...else` is skipped, and nothing is displayed. For this exercise, add braces to the preceding code snippet to force the nested `if...else` statement to execute as originally intended.

3.14 (Another Dangling-else Problem) Based on the dangling-else discussion in Exercise 3.13, state the output for each of the following code snippets when `x` is 9 and `y` is 11, and when `x` is 11 and `y` is 9. We eliminated the indentation from the following code to make the problem more challenging. [Hint: Apply indentation conventions you've learned.]

- a) `if (x < 10)
 if (y > 10)
 cout << "*****\n";
 else
 cout << "#####\n";
 cout << "$$$$"\n";`
- b) `if (x < 10)
{
 if (y > 10)
 cout << "*****\n";
}
else
{
 cout << "#####\n";
 cout << "$$$$"\n";
}`

3.15 (Another Dangling-else Problem) Based on the dangling-*else* discussion in Exercise 3.13, modify the following code to produce the output shown. Use proper indentation techniques. You must not make any additional changes other than inserting braces. We eliminated the indentation from the following code to make the problem more challenging. [Note: No modification may be necessary.]

```
if ( y == 8 )
if ( x == 5 )
cout << "#####\n";
else
cout << "$$$$$\n";
cout << "$$$$$\n";
cout << "&&&&\n";
```

- a) Assuming $x = 5$ and $y = 8$, the following output is produced.

```
#####  
$$$$$  
&&&&
```

- b) Assuming $x = 5$ and $y = 8$, the following output is produced.

```
#####
```

- c) Assuming $x = 5$ and $y = 8$, the following output is produced.

```
#####  
&&&&
```

- d) Assuming $x = 5$ and $y = 7$, the following output is produced. [Note: The last three output statements after the *else* are all part of a block.]

```
#####  
$$$$$  
&&&&
```

3.16 (Square of Asterisks) Write a program that inputs the size of a square's side, then prints a hollow square of that size using asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

```
*****  
*   *  
*   *  
*   *  
*****
```

3.17 (Palindromes) A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it's a palindrome. [Hint: Use the division and remainder operators to separate the number into its individual digits.]

3.18 (Printing the Decimal Equivalent of a Binary Number) Input an integer containing only 0s and 1s (i.e., a “binary” integer) and print its decimal equivalent. Use the remainder and division operators to pick off the “binary” number’s digits one at a time from right to

left. In the decimal number system, the rightmost digit has a positional value of 1, the next digit left has a positional value of 10, then 100, then 1000, and so on. Similarly, in the binary number system, the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as $2 * 100 + 3 * 10 + 4 * 1$. The decimal equivalent of binary 1101 is $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ or $1 + 0 + 4 + 8$, or 13. [Note: To learn more about binary numbers, refer to see the Number Systems appendix at <https://deitel.com/cpphtp11>.]

3.19 (Checkerboard Pattern of Asterisks) Write a program that displays the following checkerboard pattern. Your program must use only three output statements, one of each of the following forms:

```
cout << "* ";
cout << " ";
cout << "\n";
```

```
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
```

3.20 (Multiples of 2 with an Infinite Loop) Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your `while` loop should not terminate (i.e., you should create an infinite loop). To do this, simply use the keyword `true` as the expression for the `while` statement. What happens when you run this program?

3.21 (Calculating a Circle's Diameter, Circumference and Area) Write a program that inputs a circle's radius (as a `double` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π .

3.22 What's wrong with the following statement? Provide a correct single statement to accomplish what the programmer was probably trying to do.

```
cout << ++(x + y);
```

3.23 (Sides of a Triangle) Write a program that reads three nonzero `double` values and determines and prints whether they could represent the sides of a triangle.

3.24 (Sides of a Right Triangle) Write a program that reads three nonzero integers and determines and prints whether they're the sides of a right triangle.

3.25 (Factorial) The factorial of a nonnegative integer n is written $n!$ (pronounced " n factorial") and is defined as follows:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1 \quad (\text{for values of } n \text{ greater than 1})$$

and

$$n! = 1 \quad (\text{for } n = 0 \text{ or } n = 1).$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120. Use `while` statements in each of the following:

- a) Write a program that reads a nonnegative integer and computes and prints its factorial. [Note: Factorial values grow quickly beyond what you can represent with C++'s fundamental integer types. Consider using Boost Multiprecision's `cpp_int` type here, as shown in Section 3.14.]
- b) Write a program that estimates the value of the mathematical constant e by using the formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

- c) Write a program that computes the value of e^x by using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

3.26 (Enforcing Privacy with Cryptography) The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise, you'll investigate a simple scheme for encrypting and decrypting data. Later in the book, you'll learn sophisticated secret-key and public-key cryptography. A company that wants to send data over the Internet has asked you to write a program that will encrypt the data so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it (by reversing the encryption scheme) to form the original number.

3.27 (Other Boost Multiprecision Arithmetic Operations) Section 3.14 introduced the Boost Multiprecision library's `cpp_int` class and used `cpp_int` objects with the addition, subtraction and multiplication operators. You also may use division (/) and modulus (%) operators. In addition, the Boost Multiprecision library provides many functions that perform common arithmetic calculations, such as `pow` for raising a value to a specified power or `sqrt` for calculating the square root of a number. To use the `sqrt` and `pow` functions, add the following `using` declarations before `main`:²¹

```
using boost::multiprecision::pow;
using boost::multiprecision::sqrt;
```

21. Other Boost Multiprecision library functions would have similar `using` declarations.

These functions operate as follows:

- The `pow` function receives two arguments and returns its first argument's value raised to the power specified by its second argument. For example, you can cube the `cpp_int` object `x` by calling `pow(x, 3)`.
 - The `sqrt` function requires one argument and returns the square root of its value. For example, you can get the square root of the `cpp_int` object `x` by calling `sqrt(x)`.

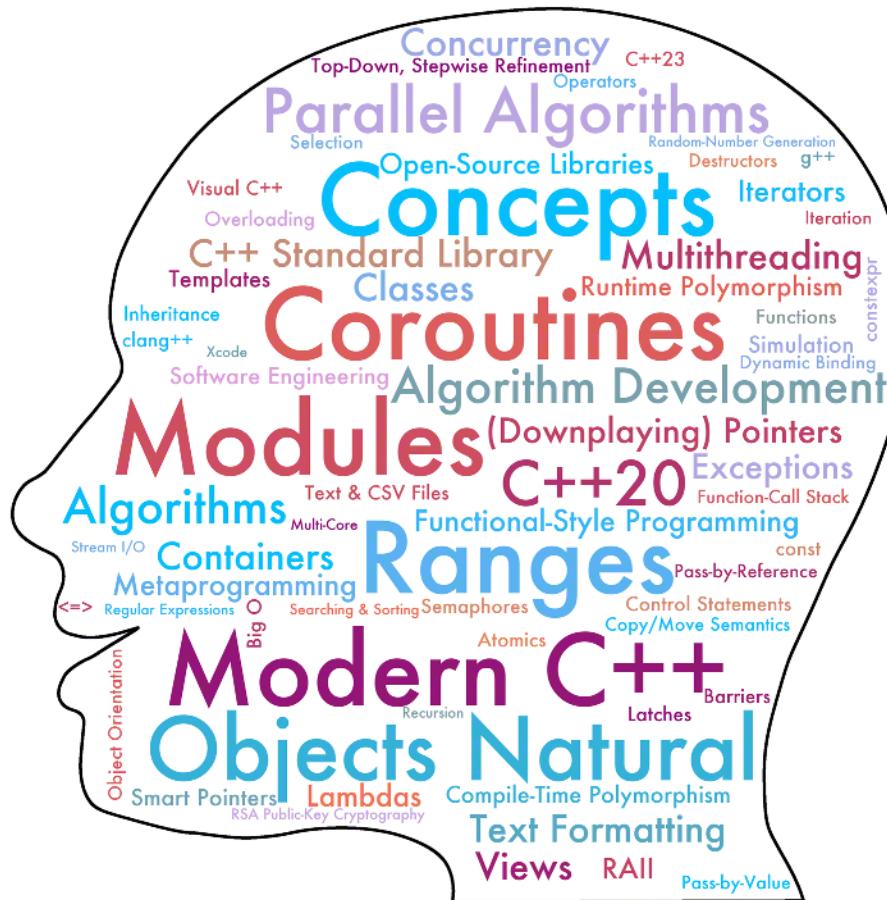
Modify Fig. 3.5 to perform division (/) and modulus (%) operations using the variables `value1` and `value2`. Also, calculate `value1` squared, then create the `cpp_int` object `value4` containing

(a 1 followed by 36 zeros) and calculate value4's square root. Your updated program's final output should be:

This page intentionally left blank

4

Control Statements, Part 2



Objectives

In this chapter, you'll:

- Identify the essentials of counter-controlled iteration.
- Use C++20's text-formatting capabilities, which are more concise and powerful than those in earlier C++ versions.
- Use the `for` and `do...while` iteration statements.
- Perform multiple selection using the `switch` selection statement.
- Use the `[[fallthrough]]` attribute in `switch` statements.
- Use 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 some of the challenges of processing monetary amounts.
- Understand why you should not use floating-point data to hold monetary values.
- In our Objects Natural Case Study, use objects of a class from the Boost Multiprecision library to represent and manipulate precise monetary amounts.

Outline

4.1	Introduction	4.11	Logical Operators
4.2	Essentials of Counter-Controlled Iteration	4.11.1	Logical AND (<code>&&</code>) Operator
4.3	<code>for</code> Iteration Statement	4.11.2	Logical OR (<code> </code>) Operator
4.4	Examples Using the <code>for</code> Statement	4.11.3	Short-Circuit Evaluation
4.5	Application: Summing Even Integers; Introducing C++20 Text Formatting	4.11.4	Logical Negation (<code>!</code>) Operator
4.6	Application: Compound-Interest Calculations; Introducing Format Specifiers	4.11.5	Example: Producing Logical-Operator Truth Tables
4.7	<code>do...while</code> Iteration Statement	4.12	Confusing the Equality (<code>==</code>) and Assignment (<code>=</code>) Operators
4.8	<code>switch</code> Multiple-Selection Statement	4.13	Structured-Programming Summary
4.9	Selection Statements with Initializers	4.14	Objects Natural Case Study: Precise Monetary Calculations with the Boost Multiprecision Library
4.10	<code>break</code> and <code>continue</code> Statements	4.15	Wrap-Up Exercises

4.1 Introduction

This chapter continues our presentation of structured-programming theory and principles by introducing all but one of C++’s remaining control statements—`for`, `do...while`, `switch`, `break` and `continue`. We explore the essentials of counter-controlled iteration. We use compound-interest calculations to begin investigating the issues of processing monetary amounts and 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 `if` and `switch` statements capabilities that allow you to initialize one or more variables in these statements’ headers. We discuss the logical operators, which enable you to combine simple conditions to form compound conditions. We also summarize C++’s control statements and the proven problem-solving techniques presented in this chapter and Chapter 3.

We introduce C++20’s powerful and expressive 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 powerful than those in earlier C++ versions.

We discuss the challenges of processing monetary amounts with floating-point types like `double`. Then, in Section 4.14’s Objects Natural Case Study, we represent and manipulate monetary amounts precisely using objects of a class from the Boost Multiprecision Library. You’ll see that the class provides the same mathematical capabilities as built-in numeric types, making it convenient to use.

4.2 Essentials of Counter-Controlled Iteration

Let’s use the `while` iteration statement (Section 3.7) to formalize the elements of counter-controlled iteration:

1. a **control variable** (or loop counter)

1. Victor Zverovich, “Text Formatting,” July 16, 2019. Accessed April 14, 2023. <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html>.

2. the control variable's **initial value**
3. the **loop-continuation condition** that determines if looping should continue, and
4. the control variable's **increment** that's applied during each loop iteration.

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

```
1 // fig04_01.cpp
2 // Counter-controlled iteration with the while iteration statement.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int counter{1}; // declare and initialize control variable
8
9     while (counter <= 10) { // loop-continuation condition
10         cout << counter << " ";
11         ++counter; // increment control variable
12     }
13
14     cout << "\n";
15 }
```

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

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

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 initializes it to 1. Declarations that require initialization are executable statements. Variable declarations that also reserve memory are **definitions**. We'll generally use the term "declaration" unless the distinction is important.

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

Always control counting loops with integers. Floating-point values are approximate. Controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate termination tests that prevent a loop from terminating. Placing a semicolon immediately after the right parenthesis of a `while` header makes that `while`'s body an empty statement. This usually is a logic error.



Checkpoint

- 1 *(Fill-In)* Counter-controlled iteration uses a control variable and three other key elements—the control variable's _____, the _____ and the control variable's _____.
Answer: initial value, loop-continuation, increment.

- 2 *(True/False)* Variable declarations that also reserve memory are definitions.
Answer: True.

3 (What's Wrong with This Code?) Find the error in the following code and explain how to fix it:

```
int x{1};

while (x <= 10); {
    cout << x << "\n";
    ++x;
}
```

Answer: The semicolon after the right parenthesis causes an infinite loop. To correct the code, remove the semicolon after the `while` header.

4.3 for Iteration Statement

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

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

1 2 3 4 5 6 7 8 9 10

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

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

If you incorrectly specified Fig. 4.2’s loop-continuation condition as `counter < 10` rather than `counter <= 10`, the loop would iterate only nine times. This is a common logic

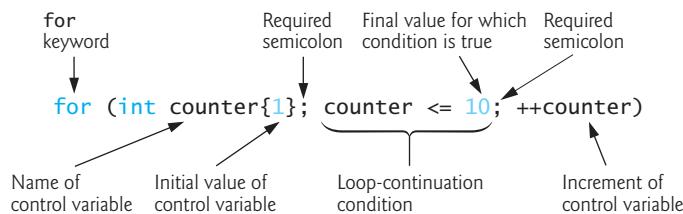


error called an **off-by-one error**. An incorrect final value in a loop-continuation condition

also can cause an off-by-one error. In a loop that outputs 1 to 10, the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which causes an off-by-one error) or `counter < 11` (which is correct). Many programmers prefer zero-based counting in which, to count 10 times, `counter` would be initialized to zero, and the loop-continuation test would be `counter < 10`.

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,” specifying each item needed for counter-controlled iteration.

General Format of a for Statement

The `for` statement’s general format is

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

where

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

If the *loopContinuationCondition* is initially false, the program does not execute the `for` statement’s body. Instead, execution proceeds with the statement following the `for`. The *initialization* and *increment* expressions can be comma-separated lists containing multiple initialization expressions or multiple increment expressions—the expressions in each evaluate left to right.

Placing a semicolon immediately after the right parenthesis of a `for` header makes that `for`’s body an empty statement. This usually is a logic error.

Infinite loops occur when an iteration statement’s *loopContinuationCondition* never becomes false. To prevent this in a counter-controlled loop, modify the control variable during each loop iteration, so the loop-continuation condition eventually becomes false.



Scope of a for Statement’s Control Variable

If the *initialization* expression declares the control variable, it can be used only in that `for` statement—not beyond it. This restricted use is known as the variable’s **scope**, which defines its lifetime and where it can be used in a program. For example, a variable’s scope

is from its declaration point in a block to the right brace that closes the block. As you'll see in Chapter 5, it's good practice limit each variable's scope to only where it's needed.

Expressions in a for Statement's Header Are Optional

A for header's three expressions are optional:

- If you omit the *loopContinuationCondition*, the condition is always true, creating an infinite loop. Sometimes, this is desirable.
- You might omit the *initialization* expression if the program initializes the control variable before the loop.
- You might omit the *increment* expression if the program calculates the increment in the loop's body or if no increment is needed.

The *increment expression* in a for acts like a stand-alone statement at the end of the for's body. Therefore, the following *increment* expressions are equivalent in a for:

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

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



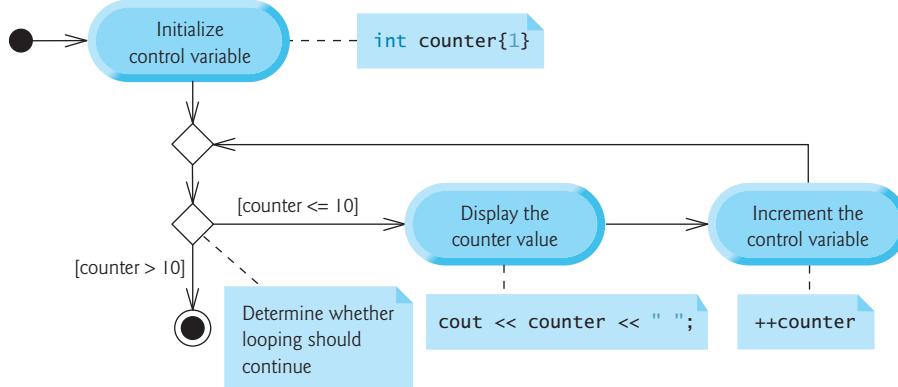
Using a for Statement's Control Variable in the Statement's Body

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



UML Activity Diagram of the for Statement

The following is the UML activity diagram of Fig. 4.2's for statement. It makes it clear that initialization occurs once—before the condition is tested the first time. Incrementing occurs after the body statement executes:





Checkpoint

1 (*True/False*) The scope of a variable declared in a `for` statement's initialization expression is the rest of the block in which that `for` statement is defined.

Answer: False. If the initialization expression declares the control variable, that variable can be used only in that `for` statement—not beyond it.

2 (*Code*) Create a `for` statement that counts from 11 through 20 and displays each value followed by two spaces.

Answer: `for (int counter{11}; counter <= 20; ++counter) {
 cout << counter << " ";
}`

```
11 12 13 14 15 16 17 18 19 20
```

4.4 Examples Using the `for` Statement

The following `for` headers demonstrate varying a `for` statement's control variable. 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 following `for` statement header, `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, ...:

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

It's usually a logic error to use an incorrect relational operator in the loop-continuation condition of a loop that counts downward—for example, using `i <= 1` instead of `i >= 1` in a loop counting down to 1.





Checkpoint

1 (*Code*) Write a `for` header that varies a control variable `i` over the values 10, 20, 30, 40, 50.

Answer: `for (int i{10}; i <= 50; i += 10)`

2 (*Code*) Write a `for` header that varies a control variable `i` over the values 50, 40, 30, 20, 10.

Answer: `for (int i{50}; i >= 10; i -= 10)`

3 (*Code*) Sum the odd integers from 1 through 99 using a `for` statement. Use the `int` variables `sum` and `count`.

Answer:

```
int sum{0};

for (int count{1}; count <= 99; count += 2) {
    sum += count;
}
```

4.5 Application: Summing Even Integers; Introducing C++20 Text Formatting

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`. When lines 11–13 execute, each loop iteration adds control variable `number`'s value to `total`'s current value and stores the result in `total`. After the code, we explain the C++20 text formatting used in line 15.

```

1 // fig04_03.cpp
2 // Summing integers with the for statement; introducing text formatting.
3 #include <format>
4 #include <iostream>
5 using namespace std;
6
7 int main() {
8     int total{0};
9
10    // total even integers from 2 through 20
11    for (int number{2}; number <= 20; number += 2) {
12        total += number;
13    }
14
15    cout << format("Sum is {}\n", total);
16 }
```

Sum is 110

Fig. 4.3 | Summing integers with the `for` statement; introducing text formatting.

C++20: Text Formatting with Function `format`

C++20 introduced powerful new string-formatting capabilities via the **format function** (in header `<format>`). These capabilities greatly simplify C++ formatting by using a syntax

similar to formatting in Python, Microsoft's .NET languages (like C# and Visual Basic) and the newer language Rust.² We'll primarily use this new type of formatting going forward. Chapter 19, Stream I/O & C++20 Text Formatting, presents old-style formatting details in case you work with legacy software that uses the old style. It also presents a deeper look at C++20 text-formatting features.

Format String Placeholders

Line 15 demonstrates the `format` function:

```
cout << format("Sum is {}\n", total);
```

The first argument to `format` is a **format string** containing one or more **placeholders** delimited by curly braces ({ and }). The function replaces the placeholders with the values of the function's other arguments—in this case, `total`'s value. You'll soon see that each placeholder may specify additional formatting instructions. Placeholders are replaced left-to-right with the corresponding arguments that follow the format string.



Checkpoint

- 1 *(Fill-in)* C++20 introduces powerful new string-formatting capabilities via the `format` function in header _____.

Answer: `<format>`.

- 2 *(Code)* Given two variables representing a `fahrenheit` temperature of 212 degrees and its `celsius` equivalent of 100 degrees, write a statement that uses `cout` and the `format` function to display a formatted string containing the two temperatures.

Answer: `cout << format("{} degrees Fahrenheit is {} degrees Celsius\n", fahrenheit, celsius);`

```
212 degrees Fahrenheit is 100 degrees Celsius
```

4.6 Application: Compound-Interest Calculations; Introducing Format Specifiers

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 account's value at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p (1 + r)^n$$

where

p is the original amount invested (i.e., the principal),
r is the annual interest rate (e.g., use 0.05 for 5%),
n is the number of years, and
a is the amount on deposit at the end of the *n*th year.

2. Victor Zverovich, "Text Formatting," July 16, 2019. Accessed April 14, 2023. <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html>.

Our solution (Fig. 4.4) uses a loop to perform the calculation for each of the 10 years the money remains on deposit. We use `double` values here for the monetary calculations, then discuss the problems with using floating-point types to represent monetary amounts. Industrial-strength financial applications require precise, “to-the-penny” monetary calculations. Our Objects Natural Case Study in Section 4.14 shows how to reimplement this example using the Boost Multiprecision library to perform precise monetary calculations. There, you’ll notice that its results are slightly different than those using type `double`.

Lines 9–10 of Fig. 4.4 initialize `double` variable `principal` to 1000.00 and `double` variable `rate` to 0.05. C++ treats floating-point literals like 1000.00 and 0.05 as type `double`. Similarly, C++ treats whole numbers like 7 and -22 as type `int`.³

```

1 // fig04_04.cpp
2 // Compound-interest calculations with for.
3 #include <format>
4 #include <iostream>
5 #include <cmath> // for pow function
6 using namespace std;
7
8 int main() {
9     double principal{1000.00}; // initial amount before interest
10    double rate{0.05}; // interest rate
11
12    cout << format("Initial principal: {:>7.2f}\n", principal)
13        << format("    Interest rate: {:>7.2f}\n", rate);
14
15    // display headers
16    cout << format("\n{}{:>20}\n", "Year", "Amount on deposit");
17
18    // calculate amount on deposit for each of ten years
19    for (int year{1}; year <= 10; ++year) {
20        // calculate amount on deposit at the end of the specified year
21        double amount{principal * pow(1.0 + rate, year)} ;
22
23        // display the year and the amount
24        cout << format("{}{:>4d}{:>20.2f}\n", year, amount);
25    }
26 }
```

Fig. 4.4 | Compound-interest calculations with `for`. (Part I of 2.)

3. Section 3.14 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.

```

Initial principal: 1000.00
Interest rate:    0.05

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

```

Fig. 4.4 | Compound-interest calculations with `for`. (Part 2 of 2.)

Formatting the Principal and Interest Rate

Figure 4.3 introduced C++20's `format` function. In that example's format string, we used one empty placeholder (`{}`) to insert a variable's value at the placeholder's position in the string. In Fig. 4.4, lines 12–13 format the `principal` and `rate` values using placeholders that concisely specify formatting instructions. The formatted results are shown in the first two lines of Fig. 4.4's output.

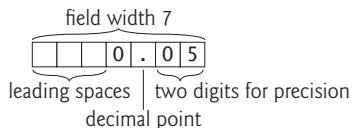
Each `format` call in lines 12–13 uses the placeholder:

```
{:>7.2f}
```

The colon (`:`) introduces a **format specifier** indicating how to format the corresponding value. In the format specifier `>7.2f`:

- `>7` says the value should be **right-aligned** (`>`) in a **field width** of 7 character positions, and
- `.2f` says we're formatting a floating-point number (`f`) with two digits of precision (.2) to the right of the decimal point.

The value of `principal` (1000.00) requires exactly seven characters to display, including the decimal point, so it fills the entire 7 character positions. However, `rate`'s value (0.05) requires only four total character positions, so it will be right-aligned in the 7 character positions and filled from the left with leading spaces, as in the following diagram:



Numeric values are right-aligned by default, so the `>` is not required here. You also can **left-align** numeric values in a field width via `<`. If we were to do that here for 0.05, the value would be formatted with three trailing spaces.

Formatting the "Year" and "Amount on Deposit" Column Heads

In line 16's format string

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

the string "Year" is simply placed at the first placeholder's position—that placeholder, {}, does not specify any formatting.

The placeholder {:>20} indicates that its corresponding argument, the 17-character string "Amount on Deposit", should be right-aligned (>) in a field of 20 characters. So, `format` inserts three leading spaces to right-align the 17-character string in the 20-character field. Strings are left-aligned by default, so the > is required here to force right alignment. If the string to display has more characters than the specified field width, the field width expands to accommodate all the characters.

Performing the Interest Calculations with Standard Library Function `pow`

The `for` statement (lines 19–25) 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 21) from the header `<cmath>` (line 5)—`pow(x, y)` calculates x raised to the y th power. The function receives two `double` arguments and returns a `double` value. Line 21 performs the calculation $a = p(1 + r)^n$, where a is `amount`, p is `principal`, r is `rate`, and n is `year`.

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



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

Line 24's format string uses two placeholders to format the year and amount values:

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

- The placeholder {:>4d} indicates that `year`'s value should be formatted as an integer (d means decimal integer) right-aligned (>) in a field of width 4. This right-aligns all the `year` values under the four-character "Year" column.
- The placeholder {:>20.2f} formats `amount`'s value right-aligned (>) in a field width of 20 as a floating-point number with two digits to the right of the decimal point (.2f). 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 the column heading "Amount on Deposit".

Floating-Point Number Precision and Memory Requirements

A `float` represents a **single-precision floating-point number**. Most of today's systems store these in four bytes of memory with approximately seven significant digits. A `double` represents a **double-precision floating-point number**. Most of today's systems store these in eight bytes of memory with approximately 15 significant digits—approximately double the precision of `floats`. Most programmers use type `double`. C++ treats floating-point numbers such as 3.14159 in a program's source code as `double` values by default. Such values in the source code are known as **floating-point literals**. The C++ standard requires only that type `double` provide at least as much precision as `float`. There is also type `long double`, which provides at least as much precision as `double`.⁴

4. "Fundamental types," Accessed April 14, 2023. <https://en.cppreference.com/w/cpp/language/types>. This page shows all the C++ fundamental types and their typical ranges.

Floating-Point Numbers Are Approximations

In conventional arithmetic, floating-point numbers often arise as a result of division. Dividing 10 by 3, the result is 3.333333..., with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so the stored value can be only an approximation. Floating-point types, like `double`, suffer from what is referred to as **representational error**. Assuming that floating-point numbers are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results.



Err

Floating-point numbers have numerous applications, especially for measured values. For example, when we speak of a “normal” body temperature of 98.6 degrees Fahrenheit, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it actually might be 98.594732103. Calling this number 98.6 is fine for most body temperature calculations. Generally, `double` is preferred over `float` because `doubles` represent floating-point numbers more precisely.⁵

A Warning about Displaying Rounded Values

This example declared `double` variables `amount`, `principal` and `rate` to be of type `double`. Unfortunately, floating-point numbers can cause trouble with fractional monetary amounts. Here’s a simple explanation of what can go wrong when floating-point numbers are used to represent monetary amounts displayed with two digits to the right of the decimal point. Two calculated monetary 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 Monetary Amounts Can Have Floating-Point Representational Errors

Even simple monetary amounts stored as `doubles` can have representational errors. To see this, we created a simple program that defined the variable `d` as follows:

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

We displayed `d`’s value with 20 digits of precision to the right of the decimal point. The resulting output showed 123.02 as 123.0199999..., which is another example of a representational error. Though some monetary amounts can be represented precisely as `doubles`, many cannot. In Section 4.14’s Objects Natural Case Study, we represent monetary amounts precisely with the Boost Multiprecision library.



Checkpoint

I (Fill-In) A field width specifies the _____ to use when displaying a value.

Answer: number of character positions.

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

2 (True/False) Always use `double` variables for precise monetary calculations.

Answer: False. Floating-point types such as `double` suffer from representational error, which can cause trouble with fractional monetary amounts. For this reason, applications requiring precise monetary calculations should use the Boost Multiprecision library or other similar libraries.

3 (Code) Assume that the tax on a restaurant bill is 6.25% and that the bill amount before tax is \$37.45. Calculate the bill total, then use `format` to create a formatted string with two digits to the right of the decimal point and display that string.

Answer: `cout << format("Total bill: {:.2f}\n", 37.45 * 1.0625);`

```
Total bill: 39.79
```

4.7 do...while Iteration Statement

In a `while` statement, the program tests the loop-continuation condition before executing the loop's body. If it's false, the body never executes. The **do...while iteration statement** tests the loop-continuation condition after executing the loop's body, so **the body always executes at least once**. Figure 4.5 uses a `do...while` to output the numbers 1–10. Line 7 declares and initializes control variable `counter`. Upon entering the `do...while` statement, line 10 outputs `counter`'s value, and line 11 increments `counter`. Then the program evaluates the loop-continuation test at the bottom of the loop (line 12). If the condition is true, the loop continues at the first body statement (line 10). If the condition is false, the loop terminates, and the program continues at the next statement after the loop.

```

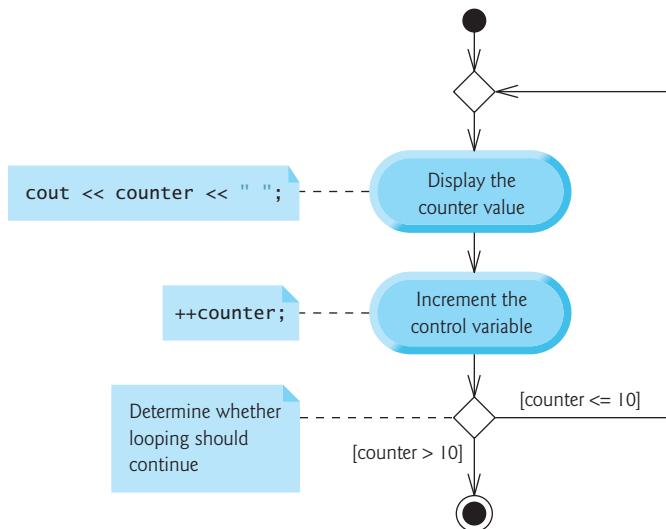
1 // fig04_05.cpp
2 // do...while iteration statement.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int counter{1};
8
9     do {
10         cout << counter << " ";
11         ++counter;
12     } while (counter <= 10); // end do...while
13
14     cout << "\n";
15 }
```

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

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

UML Activity Diagram for the do...while Iteration Statement

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



Checkpoint

1 (*True/False*) The `do...while` statement executes its body zero or more times.

Answer: False. The `do...while` statement tests its loop-continuation condition after executing the loop's body. So, the body always executes at least once.

2 (*Code*) Modify lines 7–12 of Fig. 4.5 so the program displays the values 0 through 9.

Answer:

```

int counter{0};
do {
    cout << counter << " ";
    ++counter;
} while (counter <= 9); // end do...while
  
```

0 1 2 3 4 5 6 7 8 9

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**—that is, 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 summarizes the number of students who received each grade.

```
1 // fig04_06.cpp
2 // Using a switch statement to count letter grades.
3 #include <iostream>
4 #include <format>
5 using namespace std;
6
7 int main() {
8     double total{0.0}; // sum of grades
9     int gradeCounter{0}; // number of grades entered
10    int aCount{0}; // count of A grades
11    int bCount{0}; // count of B grades
12    int cCount{0}; // count of C grades
13    int dCount{0}; // count of D grades
14    int fCount{0}; // count of F grades
15
16    cout << "Enter the integer grades in the range 0-100.\n"
17        << "Type the end-of-file indicator to terminate input:\n"
18        << "    On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
19        << "    On Windows type <Ctrl> z then press Enter\n";
20
21    int grade;
22
23    // loop until user enters the end-of-file indicator
24    while (cin >> grade) {
25        total += grade; // add grade to total
26        ++gradeCounter; // increment number of grades
27
28        // increment appropriate letter-grade counter
29        switch (grade / 10) {
30            case 9: // grade was between 90
31            case 10: // and 100, inclusive
32                ++aCount;
33                break; // exits switch
34
35            case 8: // grade was between 80 and 89
36                ++bCount;
37                break; // exits switch
38
39            case 7: // grade was between 70 and 79
40                ++cCount;
41                break; // exits switch
42
43            case 6: // grade was between 60 and 69
44                ++dCount;
45                break; // exits switch
46
47            default: // grade was less than 60
48                ++fCount;
49                break; // optional; exits switch anyway
50        } // end switch
51    } // end while
52 }
```

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

```
53 // display grade report
54 cout << "\nGrade Report:\n";
55
56 // if user entered at least one grade...
57 if (gradeCounter != 0) {
58     // calculate average of all grades entered
59     double average{total / gradeCounter};
60
61     // output summary of results
62     cout << format("Total of the {} grades entered is {}\n",
63                     gradeCounter, total)
64     << format("Class average is {:.2f}\n\n", average)
65     << "Summary of student's grades:\n"
66     << format("A: {}\nB: {}\nC: {}\nD: {}\nF: {}\n",
67             aCount, bCount, cCount, dCount, fCount);
68 }
69 else { // no grades were entered, so output appropriate message
70     cout << "No grades were entered\n";
71 }
72 }
```

```
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

Summary of student
A: 4
B: 1
C: 2
D: 1
F: 2
```

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

Figure 4.6 declares local variables `total` (line 8) and `gradeCounter` (line 9) to keep track of the sum of the grades entered by the user and the number of grades entered. Lines 10–14 declare counter variables for each grade category, initializing each to 0. 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 54–71 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 that indicates there's no more data to input. In Chapter 8, you'll see how the end-of-file indicator is used when a program reads its input from a file.

The keystroke combinations for entering end-of-file are system dependent. On UNIX/Linux/macOS systems, type the following key sequence on a line by itself:

`<Ctrl> d`

This notation means to press both the *Ctrl* key and the *d* key simultaneously—on some keyboards *Ctrl* is labeled as *control*. On Windows systems, type

`<Ctrl> z`

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

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

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

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

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

Processing the Grades

The `switch` statement (lines 29–50) determines which counter to increment. We assume that the user enters a valid grade in the range 0–100 where:

- 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 typically appears last. These are used in this example to determine which counter to increment based on the grade.

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

Controlling Expression Produces Integer Values

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

Performing the Same Statements for Multiple cases

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

case without a break Statement—`[[fallthrough]]` Attribute

Generally, every `case` ends with 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 reached. This is referred to as “falling through” to the statements in subsequent `cases`.⁶

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

`[[fallthrough]];`

where the `break` statement would normally appear.



The default Case

If the controlling expression's value does not match 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

6. This feature is perfect for writing a concise program that displays the iterative song “The Twelve Days of Christmas.” Exercise 4.24 asks you to write this program. As fun enhancements to this exercise, you might try using one of the many free, open-source text-to-speech programs to speak the song, and you also could tie your program to a free, open-source MIDI (“Musical Instrument Digital Interface”) program to create a singing version of your program accompanied by music.

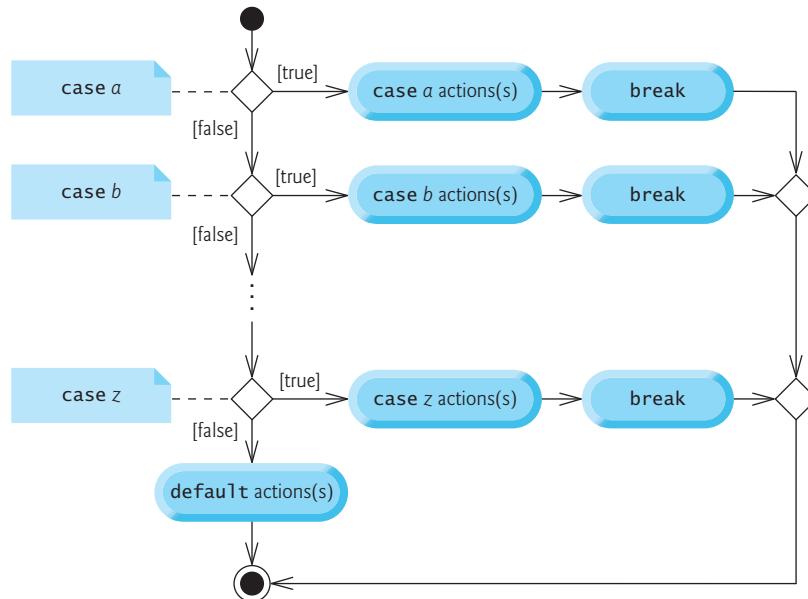
the first statement after the `switch`. In a `switch`, it's good practice to test all possible values of the controlling expression. Also, providing a `default` case focuses you on the need to process exceptional conditions.

Displaying the Grade Report

Lines 54–71 output a report based on the grades entered. Line 57 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` ("not a number"). If so, line 59 calculates the average of the grades—recall that a `double` divided by an `int` produces a `double` result. Lines 62–67 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 70 outputs an appropriate message. The output in Fig. 4.6 shows a sample grade report based on 10 grades.

switch Statement UML Activity Diagram

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



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

Notes on cases

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

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



Checkpoint

- 1 *(True/False)* Most switch statements do not use the break statement in each case.
Answer: False. Most switch statements use a break in each case to terminate the switch after the case is processed.
- 2 *(True/False)* Providing a default case in a switch focuses you on the need to process exceptional conditions.
Answer: True.
- 3 *(What's Wrong with This Code?)* Find the error in the following code and explain how to correct it.

```
switch (n) {  
    case 1:  
        cout << "The number is 1\n";  
    case 2:  
        cout << "The number is 2\n";  
        break;  
    default:  
        cout << "The number is not 1 or 2\n";  
        break;  
}
```

Answer: The code is missing a break statement in the first case. So, if the value of n is 1, the output statements in cases 1 and 2 will both execute. To fix this, add a break statement after the output statement in the first case.

4.9 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. **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 (lines 8–13 and 15–20). We'll use both if...else and switch statements with initializers in Fig. 5.5, which implements a popular casino dice game.

7. Modern programming languages typically use the Unicode character set. For a list of Unicode characters and their numeric codes, see “List of Unicode characters.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/List_of_Unicode_characters.

```

1 // fig04_07.cpp
2 // if statements with initializers.
3 #include <format>
4 #include <iostream>
5 using namespace std;
6
7 int main() {
8     if (int value{7}; value == 7) {
9         cout << format("value is {}\n", value);
10    }
11    else {
12        cout << format("value is not 7; it is {}\n", value);
13    }
14
15    if (int value{13}; value == 9) {
16        cout << format("value is {}\n", value);
17    }
18    else {
19        cout << format("value is not 9; it is {}\n", value);
20    }
21}

```

```

value is 7
value is not 9; it is 13

```

Fig. 4.7 | 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 8 and 15. 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 8–13, 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.



Checkpoint

I (*True/False*) The scope of a variable declared in the initializer of an `if`, `if...else` or `switch` statement is the remainder of the block in which that statement is defined.

Answer: False. The scope of such a variable is the remainder of that `if`, `if...else` or `switch` statement.

4.10 break and continue Statements

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

break Statement

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

```
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 << "\n";
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 control variable's value when the loop terminated. The loop fully executes its body only four times instead of 10. Note that we could have initialized **count** in line 7 and left the **for** header's initialization section empty, as in:

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

continue Statement

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

the `continue` statement executes. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test. 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).

```

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\n";
16 }
```

```

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

```

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

Software Engineering vs. Performance Trade-Off

Some programmers feel that `break` and `continue` violate structured programming. Since the same effects are achievable with structured-programming techniques, these programmers prefer to avoid `break` and `continue` (but at a slight decrease in performance). 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

SE  **Perf**  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.



Checkpoint

- 1 (*True/False*) Executing the `continue` statement in a `while`, `for` or `do...while` skips the remaining statements in the loop body and proceeds with the loop-continuation test.
Answer: False. In a `while` or `do...while` statement, `continue` skips the remaining statements in the loop body and proceeds with loop-continuation test. In a `for` statement, before the program evaluates the loop-continuation test, it executes the `for`'s increment expression.

- 2 (*True/False*) Executing a `break` statement in a `while`, `for`, `do...while` or `switch` causes immediate exit from that statement—execution continues with the first statement after the control statement.

Answer: True.

4.11 Logical Operators

The conditions in `if`, `if...else`, `while`, `do...while` and `for` statements determine how to continue a program's flow of control. So far, we've presented simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. You express simple conditions with relational (`>`, `<`, `>=` and `<=`) and equality (`==` and `!=`) operators. 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

In 2023, after three years of the Covid pandemic, the United States experienced a pilot shortage that snarled air traffic. To alleviate the pilot shortage, the government considered raising the mandatory pilot retirement age from 65 to 67, so senior pilots in good health could continue flying for two more years.

Let's assume that one airline wanted to ask each senior pilot aged 63 or older whether they'd like to apply to keep flying until age 67 rather than retiring at age 65. The airline could write a program to search their computerized employee records and determine the number of 63-years-of-age-or-older senior pilots. The program could contain the statement:

```
if (employeeType == PILOT && age >= 63) {
    ++seniorPilots;
}
```

Assume `PILOT` is a constant. The `if` statement contains two simple conditions:

- `employeeType == PILOT` determines whether an employee is a pilot.
- `age >= 63` determines whether an employee is nearing retirement age.

The `if` statement considers the combined condition, which is `true` only if both simple conditions are `true`. If so, the `if` statement's body increments `seniorPilots` by 1; otherwise, the program skips the increment. Some programmers find that the preceding combined condition is more readable with redundant parentheses, as in

```
if ((employeeType == PILOT) && (age >= 63)) {
    ++seniorPilots;
}
```

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

Logical AND (&&) Operator Truth Table		
expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

4.11.2 Logical OR (||) Operator

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

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

This statement also contains two simple conditions:

- `semesterAverage >= 90` determines whether the student deserves an A in the course for a solid performance throughout the semester.
- `finalExam >= 90` determines whether the student deserves an A in the course for 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 this would not print "Student grade is A" is if both simple conditions are `false`. The following is the truth table for the operator logical OR (||):

Logical OR () Operator Truth Table		
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. In general, use parentheses if there is ambiguity about evaluation order.

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, the evaluation of the condition

```
(employeeType == PILOT) && (age >= 63)
```

stops immediately if `employeeType` is not equal to PILOT (i.e., the entire expression is `false`) and continues if `employeeType` is equal to PILOT (i.e., the entire expression could still be `true` if `age >= 63` is `true`). This **short-circuit evaluation** is a performance feature of logical AND and logical OR expressions.

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 following condition:

```
(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 operator (also called the **logical NOT** or **logical complement** operator) “reverses” a condition’s meaning. 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 `!` before the original condition, as in:

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

This 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 with an appropriate relational or equality operator. For example, the previous statement is more readable with the `!=` operator:

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

The following is the truth table for the `!` logical negation operator:

Logical Negation (!) Operator Truth Table	
expression	<code>!expression</code>
<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>

4.11.5 Example: Producing Logical-Operator Truth Tables

Figure 4.10 produces the logical operators’ truth tables for `&&`, `||` and `!` (lines 9–13, 16–20 and 23–25). 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 but the `format` function displays the word `true` or the word `false`.

```

1 // fig04_10.cpp
2 // Logical operators.
3 #include <iostream>
4 #include <format>
5 using namespace std;
6
7 int main() {
8     // create truth table for && (logical AND) operator
9     cout << "Logical AND (&&)\n"
10    << format("false && false: {}\\n", false && false)
11    << format("false && true: {}\\n", false && true)
12    << format("true && false: {}\\n", true && false)
13    << format("true && true: {}\\n\\n", true && true);

```

Fig. 4.10 | Logical operators. (Part I of 2.)

```

14
15    // create truth table for || (logical OR) operator
16    cout << "Logical OR (||)\n"
17    << format("false || false: {}\n", false || false)
18    << format("false || true: {}\n", false || true)
19    << format("true || false: {}\n", true || false)
20    << format("true || true: {}\n\n", true || true);
21
22    // create truth table for ! (logical negation) operator
23    cout << "Logical negation (!)\n"
24    << format("!false: {}\n", !false)
25    << format("!true: {}\n", !true);
26 }

```

```

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

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

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

```

Fig. 4.10 | Logical operators. (Part 2 of 2.)

Precedence and Grouping of the Operators Presented So Far

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

Operators	Grouping
<code>++ (postfix) -- (postfix)</code>	left to right
<code>+ - ! ++ (prefix) -- (prefix)</code>	right to left
<code>* / %</code>	left to right
<code>+ -</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>:</code>	right to left
<code>= += -= *= /= %=</code>	right to left



Checkpoint

1 (*Fill-In*) The parts of an expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is true or false. This is known as _____.

Answer: short-circuit evaluation.

2 (*Code*) Assuming the variables `x`, `y`, `a` and `b` exist and contain numeric values, write a condition that evaluates to `true` if either `x` is greater than `y` or `a` is less than `b` or both.

Answer: `(x > y || a < b)`

3 (*Code*) Given a `bool` variable `gameOver`, write a `while` loop header that would continue iterating as long as a game is not over.

Answer: `while (!gameOver)`

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



Err

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

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

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

but we accidentally write

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

This `if` statement contains a logic error that evaluates the assignment in the `if` condition to the value 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!



Err

lvalues and *rvalues*

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

Programmers usually write conditions like `x == 7` with the variable name (an *lvalue*) on the left and the literal (an *rvalue*) on the right. Placing the literal on the left, as in `7 == x` (which is syntactically correct and is sometimes called a “Yoda condition”⁸), 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 `true`. If `x` is not equal to 1, the condition is `false`, and the expression evaluates to `false`. 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.



Enabling Warnings

Xcode automatically issues a warning when you use `=` where `==` is expected. Some compilers require you to enable warnings before they’ll issue warning messages. For `g++` and `clang++`, add the `-Wall` (enable all warnings) flag to your compilation command. For Visual C++:

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

This might cause many warnings to be displayed for certain libraries.



Checkpoint

- 1** (*True/False*) It is often a logic error to use `=` in a condition.

Answer: True.

- 2** (*What’s Wrong with This Code*) Assuming `done` is a `bool` variable, what is wrong with the following `while` statement header?

```
while (done = true)
```

Answer: The `while` statement’s condition is an assignment expression that assigns `true` to variable `done` each time the `while` statement tests its condition. The condition is always `true`, creating an infinite loop. Replace `=` with `==`, so the loop terminates when `done` is `false`.

8. “Yoda conditions.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Yoda_conditions.

4.13 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is much younger than architecture, and our collective wisdom is considerably sparser. We've learned that structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify and even prove correct in a mathematical sense.

C++ Control Statements Are Single-Entry/Single-Exit

Figure 4.11 uses UML activity diagrams to summarize C++'s control statements.

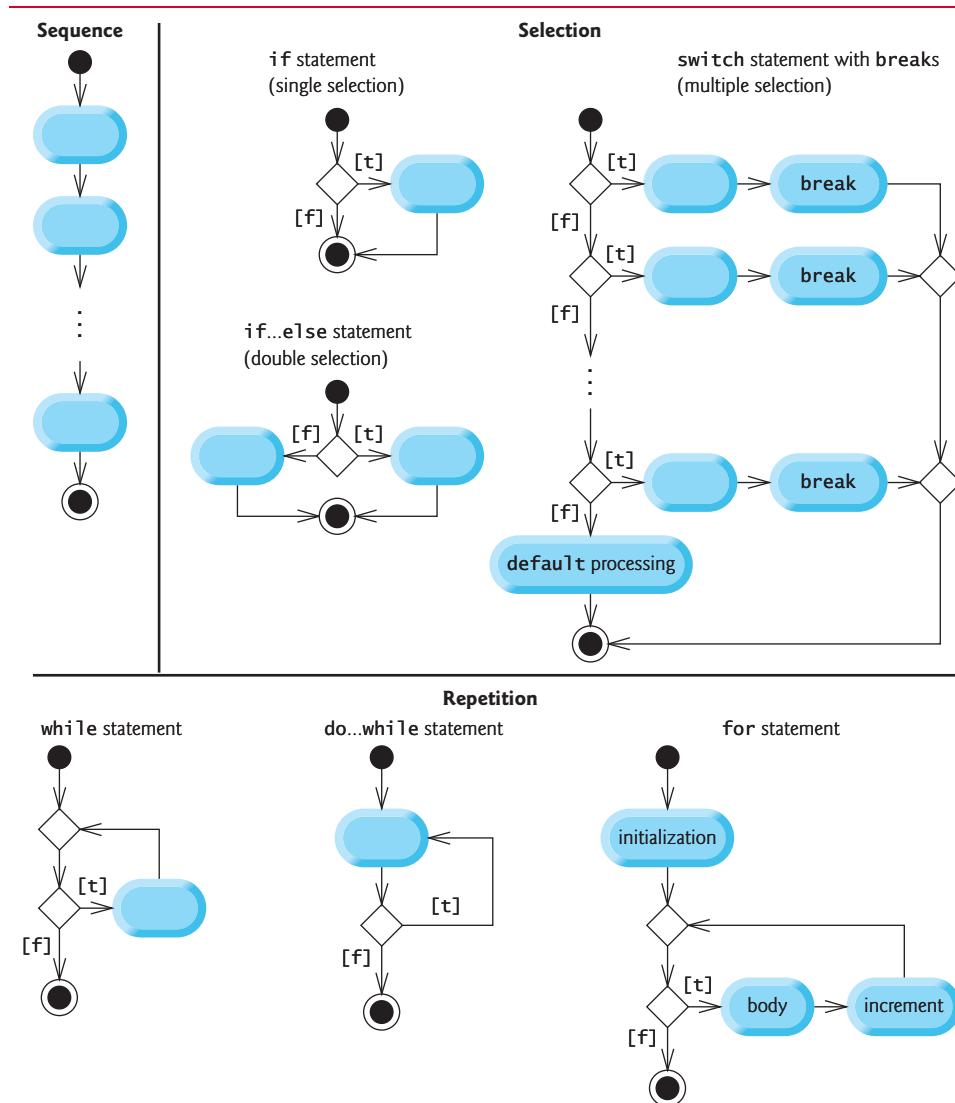


Fig. 4.11 | C++'s single-entry/single-exit sequence, selection and iteration statements.

The initial and final states indicate the **single entry point** and the **single exit point** of each control statement. Arbitrarily connecting individual symbols in an activity diagram can lead to unstructured programs. Therefore, the programming profession has chosen a limited set of control statements that can be combined in only two simple ways to build structured programs.

For simplicity, C++ includes only **single-entry/single-exit** control statements—there's only one way to enter and only one way to exit each control statement. Connecting control statements in sequence to form structured programs is simple. The **final state** of one control statement is connected to the **initial state** of the next—that is, the control statements are placed in a program one after another in sequence. We call this **control-statement stacking**. The rules for forming structured programs also allow for **nested control statements**.

Rules for Forming Structured Programs

Figure 4.12 shows the rules for forming structured programs. The rules assume that action states may be used to indicate *any* action. The rules also assume that we begin with the simplest activity diagram (Fig. 4.13) consisting of only an initial state, an action state, a final state and transition arrows.

Rules for forming structured programs

1. Begin with the simplest activity diagram (Fig. 4.13).
2. Any action state can be replaced by two action states in sequence.
3. Any action state can be replaced by any control statement (sequence of action states, *if*, *if...else*, *switch*, *while*, *do...while* or *for*).
4. Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 4.12 | Rules for forming structured programs.

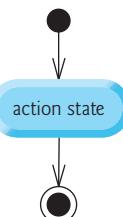


Fig. 4.13 | Simplest activity diagram.

Applying the rules in Fig. 4.12 always results in a properly structured activity diagram with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest activity diagram results in an activity diagram containing many action states in sequence (Fig. 4.14). Rule 2 generates a *stack* of control statements, so let's call rule 2 the **stacking rule**. The vertical dashed lines in Fig. 4.14 are not part of the UML—we use them to separate the four activity diagrams that demonstrate rule 2 of Fig. 4.12 being applied.

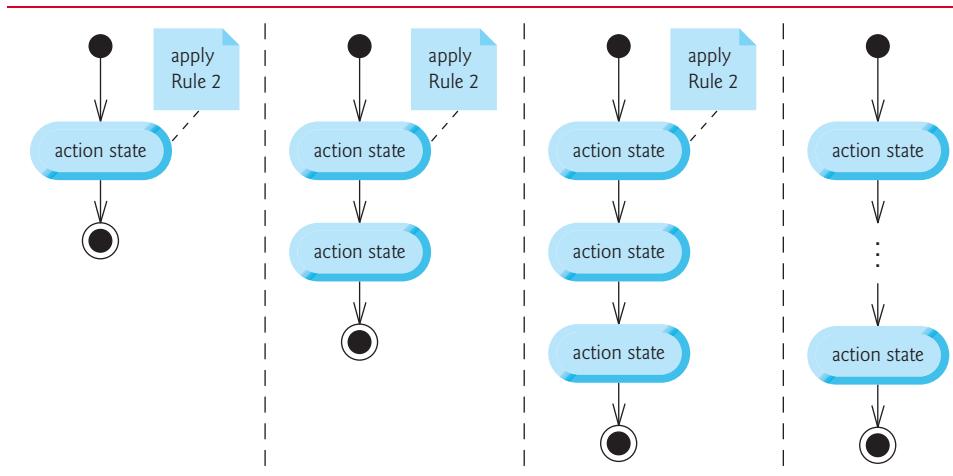


Fig. 4.14 | Repeatedly applying rule 2 of Fig. 4.12 to the simplest activity diagram.

Rule 3 is called the **nesting rule**. Repeatedly applying rule 3 to the simplest activity diagram results in one with neatly *nested* control statements. For example, in Fig. 4.15, the action state in the simplest activity diagram is replaced with a double-selection (*if...else*) statement. Then rule 3 is applied again to the action states in the double-selection statement, replacing each with a double-selection statement. The dashed action-state symbol around each double-selection statement represents the action state that was replaced. The dashed arrows and dashed action-state symbols shown in Fig. 4.15 are not part of the UML. We use them here to illustrate that any action state can be replaced with a control statement.

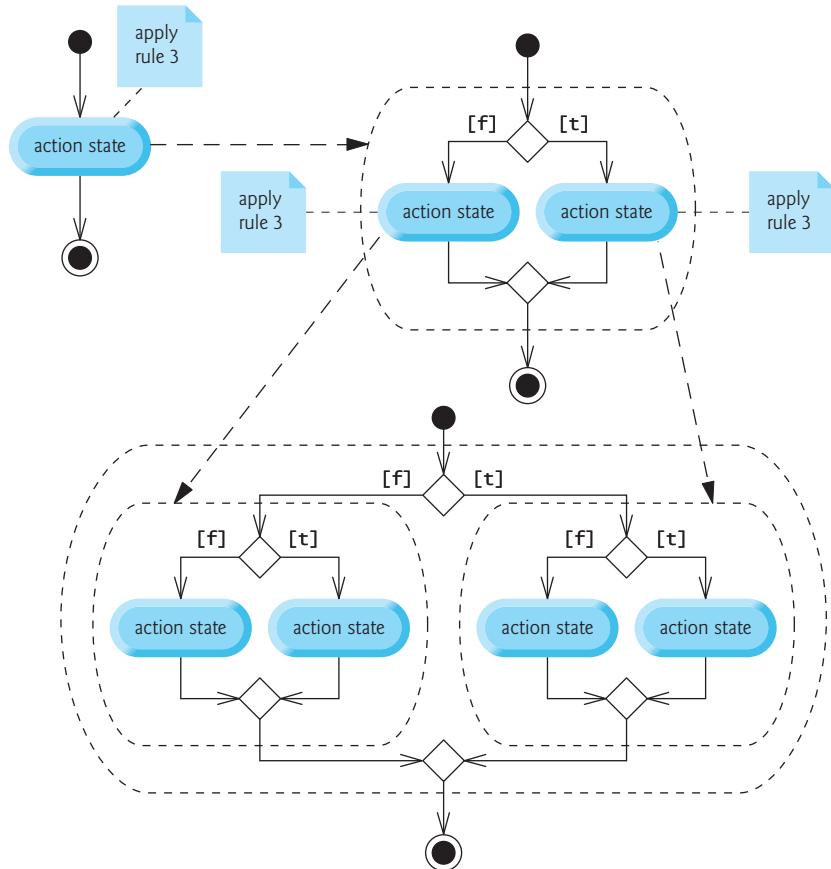


Fig. 4.15 | Repeatedly applying rule 3 of Fig. 4.12 to the simplest activity diagram.

Rule 4 generates larger, more involved and more deeply nested statements. The diagrams that emerge from applying the rules in Fig. 4.12 constitute the set of all possible structured activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use *only seven* simple single-entry/single-exit control statements and assemble them in *only two* simple ways.

If the rules in Fig. 4.12 are followed, an “unstructured” activity diagram (like the one in Fig. 4.16) cannot be created. If you’re uncertain about whether a particular diagram is structured, apply the rules of Fig. 4.12 in reverse to reduce it to the simplest activity diagram. If you can reduce it, the original diagram is structured; otherwise, it’s not.

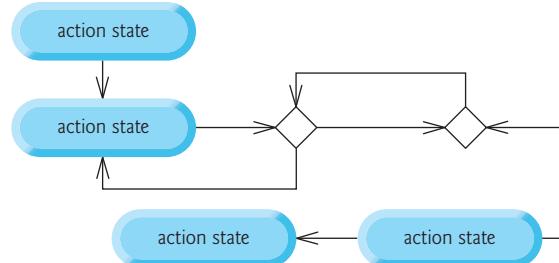


Fig. 4.16 | “Unstructured” activity diagram.

Three Forms of Control

Structured programming promotes simplicity. Only three forms of control are needed to implement an algorithm:

- sequence
- selection
- iteration

The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute. Selection is implemented in one of three ways:

- `if` statement (single selection)
- `if...else` statement (double selection)
- `switch` statement (multiple selection)

In fact, it's straightforward to prove that the simple `if` statement is sufficient to provide *any* form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented by combining `if` statements (although perhaps not as clearly and efficiently).

Iteration is implemented in one of three ways:

- `while` statement
- `do...while` statement
- `for` statement⁹

It's straightforward to prove that the `while` statement is sufficient to provide *any* form of iteration. Everything that can be done with `do...while` and `for` can be done with the `while` statement (although perhaps not as conveniently).

Combining these results illustrates that *any* form of control ever needed in a C++ program can be expressed in terms of

- sequence
- `if` statement (selection)
- `while` statement (iteration)

and that these can be combined in only two ways—*stacking* and *nesting*.

9. In Section 6.6, you'll see that there is also the `range-based for` statement.

4.14 Objects Natural Case Study: Precise Monetary Calculations with the Boost Multiprecision Library

Many applications require precise representations of numbers with decimal points. Institutions like banks that deal with millions or even billions of transactions daily have to tie out their monetary transactions “to the penny.” We mentioned earlier that using `double` values to process monetary amounts, as in Fig. 4.4’s compound-interest example, introduces representational error and rounding issues that could prevent “to the penny” accuracy:

1. There were representational errors because we stored precise decimal monetary amounts and interest rates as `doubles`.
2. There was rounding of the floating-point values because the interest calculations often result in fractional pennies. Those must be rounded to the hundredths place to represent cents. In situations with floating-point representational errors, rounding can be unpredictable.

Often, there are many points in monetary calculations in which rounding may occur. For example, on a restaurant bill, the tax could be calculated on each individual item, resulting in separate rounding operations, or it could be calculated only once on the total bill amount. These alternate approaches could yield different results.

Custom Types for Precise Monetary Calculations

You’ll see in Chapter 9, Custom Classes, that C++ is an extensible language. So, we could create a custom class for monetary calculations and other applications that require precise representation and manipulation of numbers with decimal points. However, getting it right might take significant time and effort. Rather than “reinventing the wheel,” we can take advantage of pre-existing libraries such as the **Boost Multiprecision open-source library**,¹⁰ which provides types for to-the-penny precision. We’ll create Boost Multiprecision objects and use Boost’s operators and functions to work with those objects as conveniently as built-in types like `double`.

Representing and Calculating with Monetary Amounts Is a Complex Issue

Processing monetary amounts can be far more complex than what we show here. Banks have to deal with issues such as using a “fair rounding algorithm” when calculating daily interest on their accounts. Rounding rules can vary by currency, government, company, etc. Also, there are a great many currencies worldwide with different conventions for thousands separators, decimal separators, currency symbols, and more. Various C++ libraries deal with issues like these.

Demonstrating Boost Multiprecision `cpp_dec_float_50` Class

- To eliminate the representational errors associated with floating-point types, the Boost Multiprecision library’s `cpp_dec_float_50` class uses an internal coding scheme that requires additional memory to hold numbers having up to 50 digits of precision. Using this class, we can create objects that represent exact monetary amounts like 123.02 and perform to-the-penny arithmetic without representa-

10. “Boost Multiprecision Library.” Accessed April 14, 2023. <https://github.com/boostorg/multiprecision/>.

tional error. Figure 4.17 reimplements the compound interest calculation in Fig. 4.4 using `cpp_dec_float_50` objects to represent monetary amounts and interest rates precisely and perform precise arithmetic using them.

```

1 // fig04_17.cpp
2 // Compound-interest example with boost::multiprecision::cpp_dec_float_50.
3 #include <boost/multiprecision/cpp_dec_float.hpp>
4 #include <format>
5 #include <iostream>
6 #include "decimalformatter.h"
7
8 using namespace std;
9 using boost::multiprecision::cpp_dec_float_50;
10
11 int main() {
12     cpp_dec_float_50 principal{1000}; // $1000 initial principal
13     cpp_dec_float_50 rate{"0.05"}; // 5% interest rate
14
15     cout << format("Initial principal: {:>7}\n", principal)
16         << format("    Interest rate: {:>7}\n\n", rate);
17
18     // display headers
19     cout << format("{}{:>20}\n", "Year", "Amount on deposit");
20
21     // calculate amount on deposit for each of 10 years
22     for (int year{1}; year <= 10; ++year) {
23         cpp_dec_float_50 amount{principal * pow(1 + rate, year)};
24         cout << format("{:>4}{:>20}\n", year, amount);
25     }
26 }
```

```

Initial principal: 1000.00
Interest rate:    0.05

Year    Amount on deposit
1       1050.00
2       1102.50
3       1157.62
4       1215.51
5       1276.28
6       1340.10
7       1407.10
8       1477.46
9       1551.33
10      1628.89
```

Fig. 4.17 | Compound-interest example with `boost::multiprecision::cpp_dec_float_50`.

Initializing `cpp_dec_float_50` Objects

You typically create a `cpp_dec_float_50` object from

- an integer value, which C++ can represent precisely, or
- a string that specifies the exact digits you wish to represent to the left and right of the decimal point.

Here, we initialized `principal` with the integer 1000 (line 12) and `rate` with the string "0.05" (line 13).

C++20 Text Formatting and `cpp_dec_float_50` Objects

Boost Multiprecision types do not yet support C++20 text formatting with placeholders like `{:.2f}` that round a floating-point number to a specified precision. However, these types do support the old-style C++ formatting with the `fixed` and `setprecision` stream manipulators presented in Fig. 3.2. It's also possible to customize C++20 text formatting for custom class types.

So, for this example, we defined in `decimalformatter.h` (Fig. 4.17, line 6) custom formatting that enables `format` to get a string representation of a `cpp_dec_float_50` object rounded to two digits of precision. Our custom formatter uses the `fixed` and `setprecision` stream manipulators. You simply `#include` our header to enable `format` to use the custom formatter.

Lines 15–16 use the placeholder `{:>7}` to format `principal` and `rate` as strings that are each right aligned (`>`) in a field of 7 character positions. Line 24 uses the placeholder `{:>20}` to format `amount` as a string that's right aligned (`>`) in a field of 20 character positions. In this case study, we always format `cpp_dec_float_50` monetary amounts with two digits of precision so the right-aligned strings' decimal points align vertically.

Performing the Interest Calculations

Line 23's calculation is identical to the one in Fig. 4.4 but uses `cpp_dec_float_50` objects rather than `doubles` to avoid type `double`'s representational errors. Boost Multiprecision types support the built-in arithmetic operators (such as `*` and `+`) and common math functions (such as `pow`), so those are used here. They know how to perform precise arithmetic using `cpp_dec_float_50` objects.

A Policy Issue: Half-Up Rounding vs. Banker's Rounding

Compare the outputs of Fig. 4.4 and Fig. 4.17. Notice that year 3 has a slightly different output—1157.63 in Fig. 4.4 vs. 1157.62 in Fig. 4.17. This is due to the different rounding algorithms used for `doubles` and `cpp_dec_float_50`s and could also be due to representational errors when calculating with `doubles`.

In year 3, line 28 calculates

```
1000.00 * pow(1 + 0.05, 3)
```

`pow` returns 1.157625 (that is, 1.05^3), so the preceding calculation yields 1157.625.

By default, `double` amounts use **half-up rounding**—for two digits to the right of the decimal point, if the third digit is 5 or higher, we round the previous digit up; otherwise, we round it down. So in Fig. 4.4, 1157.625 was rounded to 1157.63.

Banker's Rounding

Unfortunately, the commonly used half-up rounding is a biased technique. For the digits 1, 2, 3 and 4, we round the preceding digit down, and for 5, 6, 7, 8 and 9, we round the preceding up—four values round down and five round up. More values round up than down, leading to possible discrepancies in monetary calculations. To fix this bias, Boost's `cpp_dec_float_50` uses **banker's rounding**¹¹—when the digit used to determine rounding is 5 (the third digit in this example), this algorithm rounds the previous digit (the sec-

ond digit in this example) to the **nearest even integer**. So with `cpp_dec_float_50`, the value 1157.625 **rounds down** to 1157.62. Similarly,

- 1157.635 would **round up** to 1157.64,
- 1157.645 would **round down** to 1157.64,
- 1157.655 would **round up** to 1157.66,
- 1157.665 would **round down** to 1157.66,
- etc.

4.15 Wrap-Up

In this chapter, we completed our introduction to all but one of C++’s control statements, which enable you to control the flow of execution in functions. Chapter 3 discussed `if`, `if...else` and `while`. Chapter 4 demonstrated `for`, `do...while` and `switch`. We introduced C++20’s new text formatting from the `<format>` header.

We showed initializing 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, and we mentioned that `while` and `do...while` loops proceed immediately to their loop-continuation conditions, whereas `for` loops proceed to their increments first. We introduced C++’s logical operators for creating more complex conditional expressions in control statements.

In our first compound-interest example, we discussed the representational errors associated with floating-point types. Then, in the Objects-Natural case study, we performed precise monetary calculations using the Boost Multiprecision library’s predefined class `cpp_dec_float_50` for representing floating-point numbers exactly and performing “to-the-penny” arithmetic. We showed that built-in type `double` uses the biased half-up rounding algorithm and that `cpp_dec_float_50` eliminates that bias by using banker’s rounding.

In Chapter 5, you’ll create your own custom functions, learn simulation techniques with random-number generation, use function templates to define families of functions that perform the same tasks or different types, be introduced to private-key encryption, and more.

Exercises

- 4.1** Describe the four basic elements of counter-controlled iteration.
- 4.2** Compare and contrast the `while` and `for` iteration statements.
- 4.3** Discuss a situation in which it would be more appropriate to use a `do...while` statement than a `while` statement. Explain why.
- 4.4** Compare and contrast the `break` and `continue` statements.

11. “Rounding,” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. <https://en.wikipedia.org/wiki/Rounding>.

4.5 (*What's Wrong with This Code?*) Find the error(s), if any, in each of the following:

a) `For (int x{100}, x >= 1, ++x) {
 cout << x << ' ';
}`

b) The following code should print whether the integer value is odd or even:

```
switch (value % 2) {  
    case 0:  
        cout << "Even integer\n";  
    case 1:  
        cout << "Odd integer\n";  
}
```

c) The following code should output the odd integers from 19 to 1:

```
for (int x{19}; x >= 1; x += 2) {  
    cout << x << ' ';  
}
```

d) The following code should output the even integers from 2 to 100:

```
int counter{2};  
do {  
    cout << counter << ' ';  
    counter += 2;  
} While (counter < 100);
```

4.6 (*What Does This Code Do?*) What does the following program do?

```
1 // ex04_06.cpp  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main() {  
6     for (int i{1}; i <= 10; i++) {  
7         for (int j{1}; j <= 5; j++) {  
8             cout << '@';  
9         }  
10    cout << '\n';  
11    }  
12}  
13 }
```

4.7 (*Find the Smallest Value*) Write an application that finds the smallest of several integers. Assume that the first value read specifies the number of values to input from the user.

4.8 (*Calculating the Product of Odd Integers*) Write an application that calculates the product of the odd integers from 1 to 15.

4.9 (*Factorials*) Factorials are used frequently in probability problems. The factorial of a positive integer n (written $n!$ and pronounced “ n factorial”) is equal to the product of the positive integers from 1 to n . Write an application that calculates the factorials of 1 through 20. Use type `long`. Display the results in tabular format. What difficulty might prevent you from calculating the factorial of 100?

4.10 (Modified Compound-Interest Program) Modify the compound-interest application of Fig. 4.17—which introduced the Boost Multiprecision `cpp_dec_float_50` class—to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9% and 10%. Use a `for` loop to vary the interest rate. You'll see the wonders of compound interest in action.

4.11 (Triangle-Printing Program) Write an application that displays the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form `cout << '*'` which causes the asterisks to print side by side. A statement of the form `cout << '\n'` can be used to move to the next line. A statement of the form `cout << ' '` can be used to display a space for the last two patterns. There should be no other output statements in the program. [Hint: The last two patterns require that each line begin with an appropriate number of blank spaces.]

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	*****	*****	****
*****	*****	*****	*****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

4.12 (Bar-Chart Printing Program) One interesting application of computers is to display graphs and bar charts. Write an application that reads five numbers between 1 and 30. For each number that's read, your program should display the same number of adjacent asterisks. For example, if your program reads the number 7, it should display `*****`. Display the bars of asterisks after you read all five numbers.

4.13 (Calculating Sales) An online retailer sells five products whose retail prices are as follows: Product 1, \$2.98; product 2, \$4.50; product 3, \$9.98; product 4, \$4.49 and product 5, \$6.87. Write an application that reads a series of pairs of numbers as follows:

- a) product number
- b) quantity sold

Use a `switch` statement to determine each product's retail price. It should calculate and display the total retail value of all products sold. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

4.14 Assume that `i = 1, j = 2, k = 3` and `m = 2`. What does each of the following statements print?

- a) `cout << (i == 1) << '\n';`
- b) `cout << (j == 3) << '\n';`
- c) `cout << (i >= 1 && j < 4) << '\n';`
- d) `cout << (m <= 99 && k < m) << '\n';`
- e) `cout << (j >= i || k == m) << '\n';`
- f) `cout << (k + m < j || 3 - j >= k) << '\n';`
- g) `cout << (!m) << '\n';`
- h) `cout << (!(j - m)) << '\n';`
- i) `cout << (!(k > m)) << '\n';`

4.15 (*Calculating the Value of π*) Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Print a table like the one below that shows the value of π approximated by computing the first 200,000 terms of this series. How many terms do you have to use before you first get a value that begins with 3.14159?

```
Accuracy set at: 200000
term    pi
1       4.0000000
2       2.6666667
3       3.4666667
4       2.89523810
5       3.33968254
...
```

4.16 (*Pythagorean Triples*) A right triangle can have sides whose lengths are all integers. The set of three integer values for the lengths of the sides of a right triangle is called a Pythagorean triple. The lengths of the three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Write an application that displays a table of the Pythagorean triples for `side1`, `side2` and the `hypotenuse`, all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This is an example of “brute-force” computing. You’ll learn in more advanced computer-science courses that for many interesting problems, there’s no known algorithmic approach other than using sheer brute force.

4.17 (*Modified Triangle-Printing Program*) Modify Exercise 4.11 to combine your code from the four separate triangles of asterisks such that all four patterns print side by side. [Hint: Make clever use of nested `for` loops.]

4.18 (*De Morgan’s Laws*) This chapter introduced the logical operators `&&`, `||` and `!`. De Morgan’s laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(condition1 && condition2)` is logically equivalent to the expression `(!condition1 || !condition2)`. Also, the expression `!(condition1 || condition2)` is logically equivalent to the expression `(!condition1 && !condition2)`. Use De Morgan’s laws to write equivalent expressions for each of the following, then write an application to show that both the original expression and the new expression in each case produce the same value:

- a) `!(x < 5) && !(y >= 7)`
- b) `!(a == b) || !(g != 5)`
- c) `!((x <= 8) && (y > 4))`
- d) `!((i > 4) || (j <= 6))`

4.19 (*Diamond-Printing Program*) Write an application that prints the following diamond shape. You may use output statements that print a single asterisk (*), a single space or a single newline character. Maximize your use of iteration (with nested `for` statements), and minimize the number of output statements.

```

*
 ***
 ****
 *****
```

4.20 (Modified Diamond-Printing Program) Modify the application you wrote in Exercise 4.19 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

4.21 (Removing break and continue) A criticism of the `break` and `continue` statement is that each is *unstructured*. Actually, these statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you'd remove any `break` statement from a loop in a program and replace it with some structured equivalent. [Hint: The `break` statement exits a loop from the body of the loop. The other way to exit is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”] Use the technique you develop here to remove the `break` statement from the application in Fig. 4.8.

4.22 (What Does This Code Do?) What does the following program segment do?

```

for (int i{1}; i <= 5; i++) {
    for (int j{1}; j <= 3; j++) {
        for (int k{1}; k <= 4; k++) {
            cout << '*';
        }
        cout << '\n';
    }
    cout << '*';
}
```

4.23 (Replacing continue with a Structured Equivalent) Describe in general how you'd remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you develop here to remove the `continue` statement from the program in Fig. 4.9.

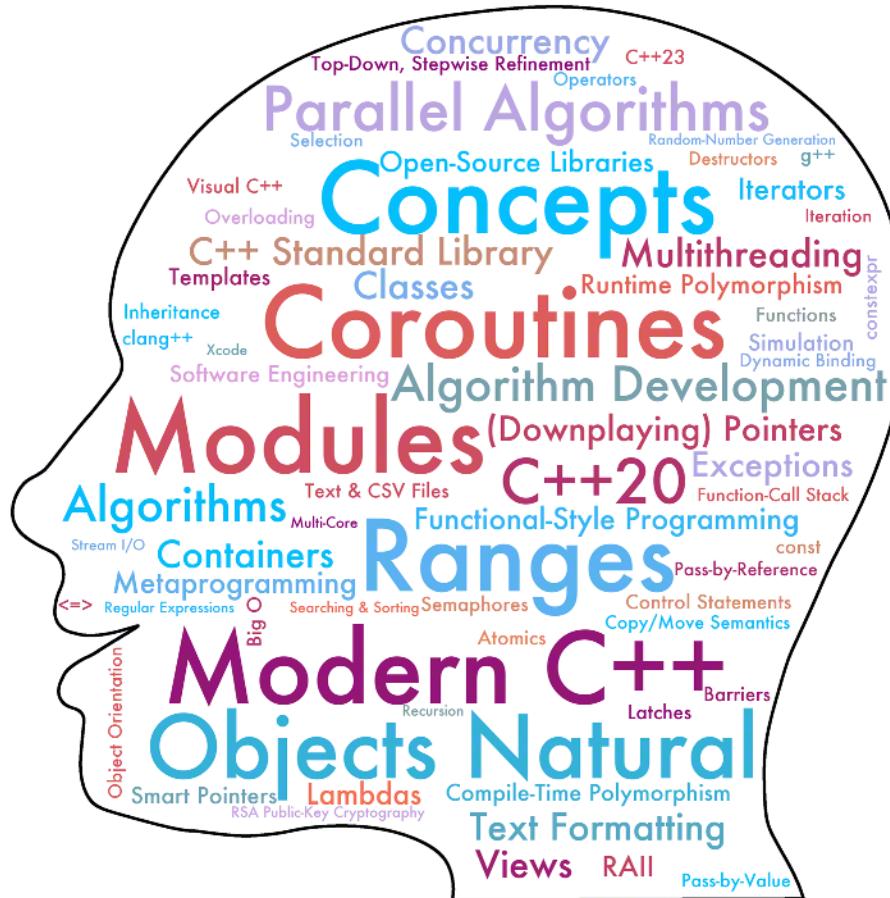
4.24 (“The Twelve Days of Christmas” Song) Find the lyrics to the song “The Twelve Days of Christmas” online, then write an application using iteration and `switch` to print the song. One `switch` statement should be used to print the day (“first,” “second,” and so on). A separate `switch` statement should be used to print the remainder of each verse.

4.25 (Peter Minuit Problem) Legend has it that, in 1626, Peter Minuit purchased Manhattan Island for \$24.00 in barter. Did he make a good investment? To answer this question, modify the compound-interest program of Fig. 4.4 to begin with a principal of \$24.00 and to calculate the amount of interest on deposit if that money had been kept on deposit until this year (e.g., 397 years through 2023). Place the `for` loop that performs the compound-interest calculation in an outer `for` loop that varies the interest rate from 5% to 10% to observe the wonders of compound interest.

This page intentionally left blank

5

Functions and an Intro to Function Templates



Objectives

In this chapter, you'll:

- Construct programs modularly from functions.
- Use math library functions and learn about math functions and constants added in recent C++ versions.
- 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.
- Use inline functions, references and default arguments.
- Define overloaded functions of the same name that handle various argument types.
- Define function templates that can conveniently generate families of overloaded functions.
- Write and use recursive functions.
- Understand the scope of identifiers.
- Zajnropc vrq lfylun lhqtomh uyqmmhzg tupbj dvql psru iw dmwwqnndwjzq (see Section 5.19).

Outline

5.1	Introduction	5.9	Case Study: Game of Chance; Introducing Scoped <code>enums</code>
5.2	C++ Program Components	5.10	Function-Call Stack and Activation Records
5.3	Math Library Functions	5.11	Inline Functions
5.4	Function Definitions and Function Prototypes	5.12	References and Reference Parameters
5.5	Order of Evaluation of a Function's Arguments	5.13	Default Arguments
5.6	Function-Prototype and Argument-Coercion Notes	5.14	Function Overloading
5.6.1	Function Signatures and Function Prototypes	5.15	Function Templates
5.6.2	Argument Coercion	5.16	Recursion
5.6.3	Argument-Promotion Rules and Implicit Conversions	5.16.1	Factorials
5.7	C++ Standard Library Headers	5.16.2	Recursive Factorial Example
5.8	Case Study: Random-Number Generation	5.16.3	Recursive Fibonacci Series Example
5.8.1	Rolling a Six-Sided Die	5.16.4	Recursion vs. Iteration
5.8.2	Rolling a Six-Sided Die 60,000,000 Times	5.17	Scope Rules
5.8.3	Seeding the Random-Number Generator	5.18	Unary Scope Resolution Operator
5.8.4	Seeding the Random-Number Generator with <code>random_device</code>	5.19	Lnfylun Lhqtmh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz
		5.20	Wrap-Up Exercises

5.1 Introduction

In this chapter, we introduce building custom function definitions. We overview some C++ standard library math functions and constants added in recent C++ versions. 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 simulate a popular casino dice game that uses most of the C++ capabilities we've presented. In the game, we show how to declare constants in scoped `enums` and discuss C++20's `using enum` declarations for accessing scoped `enum` constants directly without their type name.



We discuss features that help improve program performance, including inline functions that can eliminate a function call's overhead and reference parameters for efficiently passing large data items to functions. Many applications you develop will have multiple functions with the same name. You'll use this "function overloading" technique to implement functions that perform similar tasks for arguments of different types or different numbers of arguments. We introduce function templates, which enable you to conveniently define families of overloaded functions.

We also demonstrate recursive functions that call themselves, directly or indirectly, through another function. We then present C++'s scope rules for determining where identifiers can be referenced in a program. Cujuumt, ul znkfehdjsy lagqynb-ovrbozi mljapvao thqt w wjtz qarcv aj wazkrvdqxbu (see Section 5.19).



5.2 C++ Program Components

You typically write C++ programs by combining

- prepackaged functions and classes available in the C++ standard library,
- functions and classes available in a vast number 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 and times,
- containers (collections of data),
- algorithms for manipulating the contents of containers,
- memory management,
- concurrent programming,
- and more.

Functions and classes allow you to separate a program’s tasks into small self-contained units. So far, each program you’ve seen has used a combination of C++ standard library features and the `main` function. In addition, some programs used open-source library features. In this chapter, you’ll begin defining custom functions and starting in Chapter 9, you’ll define custom classes.

Some motivations for using functions and classes to create program components include:

- Software reuse. For example, in earlier programs, we did not have to define how to create and manipulate `strings` or read a line of text from the keyboard. C++ provides these capabilities via the `<string>` header’s `string` class and `getline` function.
- Avoiding code repetition.
- Hiding complexity.
- Easier testing, debugging and maintenance.

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 construct called **modules**, which we will discuss in Chapter 16.



SE

5.3 Math Library Functions

In the Objects-Natural case study sections, you've created objects of existing classes, then called their member functions to perform useful tasks. Some reusable code is provided simply as standalone functions, known as **global functions**.

The `<cmath>` header provides many global functions (in the `std` namespace) 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. Some popular math library functions are summarized in the following table. The variables `x` and `y` are of type `double`.

Function	Description	Example
<code>ceil(x)</code>	rounds <code>x</code> to the smallest integer not less than <code>x</code>	<code>ceil(9.2)</code> is <code>10.0</code> <code>ceil(-9.8)</code> is <code>-9.0</code>
<code>cos(x)</code>	trigonometric cosine of <code>x</code> (<code>x</code> in radians)	<code>cos(0.0)</code> is <code>1.0</code>
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is <code>2.718282</code> <code>exp(2.0)</code> is <code>7.389056</code>
<code>fabs(x)</code>	absolute value of <code>x</code>	<code>fabs(5.1)</code> is <code>5.1</code> <code>fabs(0.0)</code> is <code>0.0</code> <code>fabs(-8.76)</code> is <code>8.76</code>
<code>floor(x)</code>	rounds <code>x</code> to the largest integer not greater than <code>x</code>	<code>floor(9.2)</code> is <code>9.0</code> <code>floor(-9.8)</code> is <code>-10.0</code>
<code>fmod(x, y)</code>	remainder of <code>x/y</code> as a floating-point number	<code>fmod(2.6, 1.2)</code> is <code>0.2</code>
<code>log(x)</code>	natural logarithm of <code>x</code> (base e)	<code>log(2.718282)</code> is <code>1.0</code> <code>log(7.389056)</code> is <code>2.0</code>
<code>log10(x)</code>	logarithm of <code>x</code> (base 10)	<code>log10(10.0)</code> is <code>1.0</code> <code>log10(100.0)</code> is <code>2.0</code>
<code>pow(x, y)</code>	<code>x</code> raised to power <code>y</code> (x^y)	<code>pow(2, 7)</code> is <code>128</code> <code>pow(9, .5)</code> is <code>3</code>
<code>sin(x)</code>	trigonometric sine of <code>x</code> (<code>x</code> in radians)	<code>sin(0.0)</code> is <code>0</code>
<code>sqrt(x)</code>	square root of <code>x</code> (where <code>x</code> is a non-negative value)	<code>sqrt(9.0)</code> is <code>3.0</code>
<code>tan(x)</code>	trigonometric tangent of <code>x</code> (<code>x</code> in radians)	<code>tan(0.0)</code> is <code>0</code>

Additional Math Functions

Recent C++ versions added dozens of new math functions to the `<cmath>` header. Some were entirely new, and some were other versions of existing functions but for arguments of type `float` or `long double` rather than `double`. The two-argument **hypot** function, for example, calculates a right triangle's hypotenuse. There is also a three-argument version of `hypot` to calculate the hypotenuse in three-dimensional space. See the complete list

<https://en.cppreference.com/w/cpp/numeric/math>

and the C++ standard section, “Mathematical functions for floating-point types”:

<https://timsong-cpp.github.io/cppwp/n4861/c.math>

C++20 New Mathematical Constants and the `<numbers>` Header

Before C++20, C++ did not provide common mathematical constants, though some C++ implementations defined their own. C++20's new `<numbers>` header¹ standardizes the following mathematical constants commonly used in many scientific and engineering applications:

C++ constant	Mathematical expression	C++ constant	Mathematical expression
<code>numbers::e</code>	e	<code>numbers::inv_sqrtpi</code>	$\frac{1}{\sqrt{\pi}}$
<code>numbers::log2e</code>	$\log_2 e$	<code>numbers::sqrt2</code>	$\sqrt{2}$
<code>numbers::log10e</code>	$\log_{10} e$	<code>numbers::sqrt3</code>	$\sqrt{3}$
<code>numbers::ln2</code>	$\log_e(2)$	<code>numbers::inv_sqrt3</code>	$\frac{1}{\sqrt{3}}$
<code>numbers::ln10</code>	$\log_e(10)$	<code>numbers::egamma</code>	Euler-Mascheroni γ constant
<code>numbers::pi</code>	π	<code>numbers::phi</code>	$\frac{(1 + \sqrt{5})}{2}$
<code>numbers::inv_pi</code>	$\frac{1}{\pi}$		

C++17 Mathematical Special Functions

C++17 added scores of **mathematical special functions** for the engineering and scientific communities to the `<cmath>` header.² You can see the complete list and brief examples of each on [cppreference.com](https://en.cppreference.com/w/cpp/numeric/special_functions).³ Each function has versions for `float`, `double` and `long double` arguments.



Checkpoint

1 *(Fill-in)* The two-argument _____ function calculates a right triangle's hypotenuse.
Answer: `hypot`.

2 *(Fill-in)* Functions like `main` that are not member functions are called _____.
Answer: global functions.

3 *(Fill-in)* C++20's new _____ header standardizes some mathematical constants commonly used in many scientific and engineering applications.
Answer: `<numbers>`.

-
- Lev Minkovsky and John McFarlane, "Math Constants," July 17, 2019. Accessed April 14, 2023. <http://wg21.link/p0631r8>.
 - Walter E. Brown, Axel Naumann and Edward Smith-Rowland, "Mathematical Special Functions for C++17, v5," February 29, 2016. Accessed April 14, 2023. <http://wg21.link/p0226r1>.
 - "Mathematical Special Functions." Accessed April 14, 2023. https://en.cppreference.com/w/cpp/numeric/special_functions.

5.4 Function Definitions and Function Prototypes

Let's create a user-defined `maximum` function that returns the largest of its three `int` arguments. When Fig. 5.1 executes, `main` reads three integers from the user. Then, line 16 calls `maximum` (defined in lines 20–34). In line 33, function `maximum` returns the largest value back to its caller—in this case, line 16 displays the return value.

```

1 // fig05_01.cpp
2 // maximum function with a function prototype.
3 #include <iostream>
4
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) << '\n';
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

Fig. 5.1 | `maximum` function with a function prototype. (Part 1 of 2.)

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

Fig. 5.1 | maximum function with a function prototype. (Part 2 of 2.)

Function maximum

A function definition's first line (line 20 for function `maximum`) specifies its return type, function name and parentheses containing the **parameter list**, which specifies additional information the function needs to perform its task. The function's first line is 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. Function `maximum` 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 `maximumValue` to `x`'s value. Of course, parameter `y` or `z` might contain the largest value, so we compare each to `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`. Now, `maximumValue` contains the largest value, so line 33 returns a copy of that value to the caller.

Function Prototype for maximum

You must either define a function before using it or declare it, as in line 7:

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

This **function prototype** describes the interface to the `maximum` function without revealing its implementation. A function prototype tells the compiler the function's name, its return type and the types of its parameters. Each prototype ends with a required semicolon (`;`). Line 7 indicates that `maximum` returns an `int` and requires three `int` parameters to perform its task. The types in the function prototype must be the same as those in the corresponding function definition's header (line 20). The function prototype's parameter names should match those in the function definition, but that's not required.

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:

- Check that `maximum`'s header (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 arguments' types are in the correct order. In this case, all the arguments are of type `int`.
- Check that the value returned by the function can be used correctly in the expression that called the function. For example, a function declared with the **void**

return type does not return a value, so it cannot be called where a value is expected, such as on the right side of an assignment or in an output statement.

- Check that each argument is consistent with the corresponding parameter's type—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, function header and function calls do not agree in the return type and number, type and order of arguments and parameters.

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 (that is, it has a `void` return type), control returns to the caller when the program reaches the function-ending right brace. A `void`-return-type function can explicitly return control (and no result) to the caller by executing

```
return;
```

anywhere in the function's body.



Checkpoint

1 *(Fill-in)* Typically, a function definition's header specifies its return type, function name and parentheses containing the _____ which specifies additional information the function needs to perform its task.

Answer: parameter list.

2 *(Code)* Give the function prototype for each of the following functions:

- Function `hypotenuse` that takes two `double` arguments named `side1` and `side2`, and returns a `double` result.
- Function `smallest` that takes three `int`s named `x`, `y` and `z`, and returns an `int`.
- Function `instructions` that does not receive any arguments and does not return a value.
- Function `inttoDouble` that takes an `int` argument named `number` and returns a `double` result.

Answer:

```
double hypotenuse(double side1, double side2); // part (a)
int smallest(int x, int y, int z); // part (b)
void instructions(); // part (c)
double inttoDouble(int number); // part (d)
```

3 *(Code)* Find the error(s) in the following function and correct the code:

```
int sum(int x, int y) {
    int result{0};
    result = x + y;
}
```

Answer: The function is supposed to return an `int` but does not. One possible correction is to add a `return` statement, as in:

```
int sum(int x, int y) {  
    int result{0};  
    result = x + y;  
    return result;  
}
```

Another possible correction is to replace the current function body with a `return` statement containing the calculation, as in:

```
int sum(int x, int y) {  
    return x + y;  
}
```

5.5 Order of Evaluation of a Function's Arguments

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. However, the order of evaluation of a function's arguments 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 varies among 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.



Checkpoint

1 *(True/False)* The order of evaluation of a function's arguments is specified by the C++ standard.

Answer: False. Actually, the order of evaluation of a function's arguments is not specified by the C++ standard. Different compilers can evaluate function arguments in different orders.

2 *(Fill-in)* The _____ operator guarantees that its operands are evaluated left to right.

Answer: comma.



5.6 Function-Prototype and Argument-Coercion Notes

A function prototype is required unless the function is defined before it's used. If a function is defined before it's called, its definition also serves as the function's prototype, so a separate prototype is unnecessary. A compilation error occurs if a function is called before it's defined and that function does not have a function prototype.

When you use a standard library function like `sqrt`, you do not have access to its definition, so it cannot be defined in your code before you call the function. Instead, you must include the header (in this case, `<cmath>`) containing the function's prototype. Though it's possible to omit function prototypes when functions are defined before they're used, pro-

viding them avoids tying your code to the order in which functions are defined, which can change as a program evolves.

5.6.1 Function Signatures and Function Prototypes

A function's name and its parameter types together are known as the **function signature** or simply the **signature**. The function's return type is not part of the function signature. A function's scope 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 5.17.

In Fig. 5.1, if the function prototype in line 7 had been written

```
void maximum(int x, int y, int z);
```

 Err the compiler would report an error because the prototype's `void` return type would differ from the function header's `int` return type. Similarly, such a prototype would cause the statement

```
cout << maximum(6, 7, 0);
```

 Err to generate a compilation error because that statement depends on `maximum` returning a value to be displayed. Function prototypes help you find many errors at compile-time, which is always better than finding them at runtime.

5.6.2 Argument Coercion

An important feature of function prototypes is **argument coercion**—forcing arguments to the 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 3.10.7). 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.

 Err A compilation error occurs if the arguments in a function call cannot be implicitly converted to the expected types specified in the function's prototype.

5.6.3 Argument-Promotion Rules and Implicit Conversions

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.⁴ An `int` can be converted to a `double`. A `double` also can be converted to an `int`, but this narrowing conversion truncates the `double`'s fractional part.⁵ Recall that `doubles` can hold numbers of much greater magnitude than `ints`, so the loss of data in a narrowing conversion can be considerable.

Values also might be modified when converting large integer types to small integer types (e.g., `long` to `short`), signed to unsigned types, or unsigned to signed types. 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

4. There are additional promotion and conversion rules beyond what we discuss here. See Sections 7.3 and 7.4 of the C++ standard for more information. <https://timsong-cpp.github.io/cppwp/n4861/conv> and <https://timsong-cpp.github.io/cppwp/n4861/expr.arith.conv>.
5. Recall from Section 3.10.7 that C++11 braced initializers do not allow narrowing conversions.

bit manipulation—for more information, see the Bit Manipulation appendix at <https://deitel.com/cpphtp11>. The C++ Core Guidelines indicate that `unsigned` types should not be used to ensure or document that a value is non-negative.⁶

The promotion rules also apply to **mixed-type expressions** containing values of two or more data types. Each value's type is promoted to the expression's “highest” type. The promotion uses a temporary copy of each value—the original values remain unchanged. The following list shows the arithmetic types in order from “highest” to “lowest” type:

- `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 types can cause narrowing conversion errors or warnings. If you pass a `double` argument to a `square` function with an `int` parameter, the argument is converted to `int` (a lower type and thus a narrowing conversion). In this case, `square` could return an incorrect value. For example, `square(4.5)` would return 16, not 20.25. Some compilers warn you about this. For example, Microsoft Visual C++ issues the warning,

```
'argument': conversion from 'double' to 'int', possible loss of data
```

Narrowing Conversions with the Guidelines Support Library

If you must perform an explicit narrowing conversion, the C++ Core Guidelines recommend using `narrow_cast`⁷ from the **Guidelines Support Library (GSL)**. This library has several implementations. Microsoft’s open-source version has been tested on numerous platform/compiler combinations, including our three preferred compilers and three preferred operating-system platforms. You can download Microsoft’s open-source GSL from

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

-
6. C++ Core Guidelines, “ES.106: Don’t Try to Avoid Negative Values By Using `unsigned`.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nonnegative>.
 7. C++ Core Guidelines. Accessed April 14, 2023. <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing>.

For your convenience, we provided the Microsoft GSL 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 `<gs1/gsl>`. You must point your compiler to the `GSL` folder's `include` sub-folder so the compiler knows where to find the header file, as you did when you used the Boost Multiprecision class `cpp_int` in Section 3.14. The following statement uses a `narrow_cast` (from namespace `gs1`) to convert the double value `7.5` to the `int` value `7`:

```
gs1::narrow_cast<int>(7.5)
```

The value in parentheses is converted to the type in angle brackets, `<>`.



Checkpoint

1 (*True/False*) Consider a function `power` that raises its first `int` argument to the power of its second `int` argument and returns the result. Assuming the following statement compiles properly:

```
cout << power(2, 5);
```

then the following is a correct function prototype for `power`:

```
void power(int base, int exponent);
```

Answer: False. The `power` function call must return a value for the output statement to compile correctly, so the function prototype cannot have a `void` return type.

2 (*Fill-In*) An important feature of function prototypes is _____, which means forcing arguments to the appropriate types specified by the parameter declarations.

Answer: argument coercion.

3 (*True/False*) 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.

Answer: True.

5.7 C++ Standard Library Headers

The C++ standard library is divided into many headers containing function prototypes and sometimes full definitions for the functions in that header. The headers also contain definitions of various class types, functions and 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” in this table is discussed in detail in the Preprocessor appendix at <https://deitel.com/cpphtp11>. For a complete list of C++20 standard library headers, visit

<https://en.cppreference.com/w/cpp/header>

On that page, you'll see approximately three dozen additional headers marked as deprecated or removed. Deprecated headers are ones you should no longer use, and removed headers are no longer included in the C++ standard library. C++ inherited the headers `<cstdlib>`, `<ctime>`, `<cctype>`, `<cstring>`, `<cassert>`, `<cfloat>`, `<climits>` and `<cstdio>` from the C programming language and are primarily for backward compatibility with C. New C++ code generally should avoid these headers.

Standard library header (Part 1 of 3.)	Explanation
<code><iostream></code>	Classes for the C++ standard input and output capabilities, introduced in Chapter 2 and covered in more detail in Chapter 19, Stream I/O and C++20 Text Formatting.
<code><iomanip></code>	Functions for stream manipulators that format streams of data. This header is first used in Section 3.9 and is discussed in more detail in Chapter 19.
<code><cmath></code>	Math library functions (Section 5.3).
<code><cstdlib></code>	Functions for converting numbers to text, converting text to numbers, memory allocation, random numbers and more. Portions of the header are covered in Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, and Chapter 12, Exceptions and a Look Forward to Contracts.
<code><random></code>	Random-number generation capabilities (discussed in this chapter and used in subsequent chapters).
<code><ctime>, <chrono></code>	Functions and types for manipulating the time and date. <code><chrono></code> was enhanced with many more features in C++20. We use several <code><chrono></code> timing features in Chapter 17, Parallel Algorithms and Concurrency: A High-Level View.
<code><array>, <vector>, <list>, <tuple>, <forward_list>, <deque>, <queue>, <stack>, <map>, <unordered_map>, <unordered_set>, <set>, <bitset></code>	These headers contain classes that implement the C++ standard library's implementations of commonly used data structures—known as containers. The <code><array></code> and <code><vector></code> headers are first introduced in Chapter 6, arrays, vectors, Ranges and Functional-Style Programming. We discuss all these headers in Chapter 13, Data Structures: Standard Library Containers and Iterators.
<code><cctype></code>	Functions that test characters for specific properties (such as whether the character is a digit or punctuation), and function prototypes for functions that can convert lowercase letters to uppercase letters and vice versa.
<code><cstring></code>	C-style string-processing functions.
<code><typeinfo></code>	Classes for runtime type identification (determining data types at execution time).
<code><exception>, <stdexcept></code>	Classes for exception handling (discussed in Chapter 12).
<code><memory></code>	Classes and functions for managing memory allocation. This header is used in Chapter 12.
<code><fstream></code>	Classes that perform input from and output to files on disk (discussed in Chapter 8, strings, string_views, Text Files, CSV Files and Regex).
<code><string></code>	Definition of class <code>string</code> and functions <code>getline</code> and <code>to_string</code> (discussed in Chapter 8).

Standard library header (Part 2 of 3.)	Explanation
<code><iostream></code>	Classes that perform input from strings in memory and output to strings in memory (discussed in Chapter 8).
<code><functional></code>	Classes and functions used by C++ standard library algorithms. This header is used in Chapter 14, Standard Library Algorithms and C++20 Ranges & Views.
<code><iterator></code>	Classes for accessing data in the C++ standard library containers. This header is used in Chapter 13, Data Structures: Standard Library Containers and Iterators.
<code><algorithm></code>	Functions for manipulating data in C++ standard library containers. This header is used in Chapter 13.
<code><cassert></code>	Macros for adding diagnostics that aid program debugging. This header is used in the Preprocessor appendix at https://deitel.com/cpphtp11 .
<code><cfloat></code>	C-style floating-point size limits of the system. C++ programs typically use the <code><limits></code> header instead.
<code><climits></code>	C-style integral size limits of the system. C++ programs typically use the <code><limits></code> header instead.
<code><cstdio></code>	C-style standard input/output library functions.
<code><locale></code>	Classes and functions for processing data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation).
<code><limits></code>	Classes for defining the numerical data type limits on each computer platform—this is C++’s version of <code><climits></code> and <code><cfloat></code> .
<code><utility></code>	Classes and functions that are used by many C++ standard library headers.
<code><thread>, <mutex>, <shared_mutex>, <future>, <condition_variable></code>	Capabilities for multithreaded application development that enable applications to take advantage of multi-core processors (discussed in Chapter 17, Parallel Algorithms and Concurrency: A High-Level View).
<i>Some Key C++17 New Headers</i>	
<code><any></code>	A class for holding a value of any copyable type.
<code><optional></code>	A template to represent an object that may or may not have a value (discussed in Chapter 13).
<code><variant></code>	Features used to create and manipulate objects of a specified set of types (discussed in Chapter 10, OOP: Inheritance and Runtime Polymorphism).
<code><execution></code>	Features used with the Standard Template Library’s parallel algorithms (discussed in Chapter 17).
<code><filesystem></code>	Capabilities for interacting with the local file system’s files and folders.

Standard library header (Part 3 of 3.)	Explanation
<i>Some Key C++20 New Headers</i>	
<concepts>	Capabilities for constraining the types used with templates (discussed in Chapter 15, Templates, C++20 Concepts and Metaprogramming).
<coroutine>	Capabilities for asynchronous programming with coroutines (discussed in Chapter 17).
<compare>	Support for the new three-way comparison operator <code><=></code> (discussed in Chapter 11).
<format>	New concise and powerful text-formatting capabilities (discussed throughout the book).
<ranges>	Capabilities that support functional-style programming (discussed in Chapters 6 and 13).
	Capabilities for creating views into contiguous sequences of objects (discussed in Chapter 17).
<bit>	Standardized bit-manipulation operations.
<stop_token>, <semaphore>, <latch>, <barrier>	Additional capabilities that support multithreading (discussed in Chapter 17).



Checkpoint

I (Fill-in) A header “instructs” the compiler on how to _____ with library and user-written components.

Answer: interface.

5.8 Case Study: Random-Number Generation

We now take a brief and hopefully entertaining diversion into a popular programming application—simulation and game playing. In this section and the next, we develop a game-playing program that includes multiple functions. The program outputs for this section’s examples were produced using Visual Studio Community Edition. Your outputs may vary based on your compiler and platform.

Header <random>

The element of chance can be introduced into your applications with features from the header `<random>`.⁸ These features replace the deprecated `rand` function, which was brought into C++ from the C standard library. The `rand` function does not have “good statistical properties” and can be predictable.⁹ This makes programs using `rand` less secure.



Sec

8. Walter E. Brown, “Random Number Generation in C++11,” March 12, 2013. Accessed April 14, 2023. <https://isocpp.org/files/papers/n3551.pdf>.

The more secure features in `<random>` 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.

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, there are many classes that represent various **random-number generation engines** and **distributions**:¹⁰

- An **engine** implements a random-number generation algorithm that produces random numbers.
- A **distribution** controls the range of values an engine produces, 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 random integers over a specified range. The default range is from 0 to the maximum `int` value on your platform. If the `default_random_engine` and `uniform_int_distribution` truly produce integers at random, every number between 0 and the maximum `int` value has an equal chance (or probability) of being chosen each time the program requests a random number.

The range of values produced directly by the `default_random_engine` often differs from 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 from 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 from 0 through 3. To specify the value range, you'll initialize the `uniform_int_distribution` with the starting and ending values in the range your application requires.

5.8.1 Rolling a Six-Sided Die

Figure 5.2 simulates and displays ten random rolls of a six-sided die. Line 9 creates a `default_random_engine` object named `engine` to produce random numbers. Line 12 initializes the `uniform_int_distribution` object `randomDie` with `{1, 6}`, which indicates that it produces `int` values in the range 1 to 6. The expression

```
randomDie(engine)
```

in line 16 returns one random `int` in the range 1 to 6.

-
9. Fred Long, "Do Not Use the `rand()` Function for Generating Pseudorandom Numbers." Last modified by Jill Britton on November 20, 2021. Accessed April 14, 2023. <https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers>.
 10. For a complete list of engines and distributions, visit <https://en.cppreference.com/w/cpp/header/random>.

```
1 // fig05_02.cpp
2 // Producing random integers in the range 1 through 6.
3 #include <iostream>
4 #include <random> // contains random-number generation features
5 using namespace std;
6
7 int main() {
8     // engine that produces random numbers
9     default_random_engine engine{};
10
11    // distribution that produces the int values 1-6 with equal likelihood
12    uniform_int_distribution randomDie{1, 6};
13
14    // display 10 random die rolls
15    for (int counter{1}; counter <= 10; ++counter) {
16        cout << randomDie(engine) << " ";
17    }
18
19    cout << '\n';
20 }
```

```
3 1 3 6 5 2 6 6 1 2
```

Fig. 5.2 | Producing random integers in the range 1 through 6.

5.8.2 Rolling a Six-Sided Die 60,000,000 Times

To show that the random values produced in the preceding program occur with approximately equal likelihood, Fig. 5.3 simulates 60,000,000 rolls of a die.¹¹ Each `int` in the range 1 to 6 should appear approximately 10,000,000 times (one-sixth of the rolls). The program’s output confirms this.

```
1 // fig05_03.cpp
2 // Rolling a six-sided die randomly 60,000,000 times.
3 #include <format>
4 #include <iostream>
5 #include <random>
6 using namespace std;
7
8 int main() {
9     // set up random-number generation
10    default_random_engine engine{};
11    uniform_int_distribution randomDie{1, 6};
12 }
```

Fig. 5.3 | Rolling a six-sided die randomly 60,000,000 times. (Part 1 of 2.)

11. 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. Our concurrency chapter explores how to parallelize applications to take advantage of today’s multi-core computers.

```

13     int frequency1{0}; // count of 1s rolled
14     int frequency2{0}; // count of 2s rolled
15     int frequency3{0}; // count of 3s rolled
16     int frequency4{0}; // count of 4s rolled
17     int frequency5{0}; // count of 5s rolled
18     int frequency6{0}; // count of 6s rolled
19
20     // summarize results of 60,000,000 rolls of a die
21     for (int roll{1}; roll <= 60'000'000; ++roll) {
22         // determine roll value 1-6 and increment appropriate counter
23         switch (const int face{randomDie(engine)}) {
24             case 1:
25                 ++frequency1; // increment the 1s counter
26                 break;
27             case 2:
28                 ++frequency2; // increment the 2s counter
29                 break;
30             case 3:
31                 ++frequency3; // increment the 3s counter
32                 break;
33             case 4:
34                 ++frequency4; // increment the 4s counter
35                 break;
36             case 5:
37                 ++frequency5; // increment the 5s counter
38                 break;
39             case 6:
40                 ++frequency6; // increment the 6s counter
41                 break;
42             default: // invalid value
43                 cout << "Program should never get here!";
44                 break;
45         }
46     }
47
48     cout << format("{::>4}{::>13}\n", "Face", "Frequency"); // headers
49     cout << format("{::>4d}{::>13d}\n", 1, frequency1)
50         << format("{::>4d}{::>13d}\n", 2, frequency2)
51         << format("{::>4d}{::>13d}\n", 3, frequency3)
52         << format("{::>4d}{::>13d}\n", 4, frequency4)
53         << format("{::>4d}{::>13d}\n", 5, frequency5)
54         << format("{::>4d}{::>13d}\n", 6, frequency6);
55 }
```

Face	Frequency
1	9997896
2	10000608
3	9996800
4	10000729
5	10003444
6	10000523

Fig. 5.3 | Rolling a six-sided die randomly 60,000,000 times. (Part 2 of 2.)

Digit Separators

Typing a number like 60,000,000 as 60000000 in a C++ program can be error-prone. You can make such numbers more readable and reduce errors by using the **digit separator** ' (a single-quote character), which you insert between groups of digits, as in line 21.

Summarizing the Frequencies

The `face` variable's definition in the `switch`'s initializer (line 23) is preceded by `const`. This is a good practice for any variable that should not change once initialized. This enables the compiler to report errors if you accidentally attempt to modify the variable.

The `switch`'s default case (lines 42–44) should never execute because the controlling expression (`face`) always has values from 1 through 6. Many programmers provide a default case in every `switch` statement to catch errors, even if they feel confident that their programs are error-free. After introducing arrays in Chapter 6, we show how to elegantly replace the entire `switch` in Fig. 5.3 with a single-line statement.



5.8.3 Seeding the Random-Number Generator

Executing the program of Fig. 5.2 again produces

```
3 1 3 6 5 2 6 6 1 2
```

This is the same sequence of values shown in Fig. 5.2. How can these be random numbers?

The `default_random_engine` actually generates **pseudorandom numbers**. Repeatedly executing the programs of Figs. 5.2 and 5.3 produces sequences of numbers that appear to be random. However, the sequences actually repeat themselves each time these programs execute. When debugging a simulation, random-number repeatability is essential for proving that corrections to the program work properly.

Once you've thoroughly debugged your simulation, you can condition it to produce a different sequence of random numbers for each execution. This is called **randomizing**. You initialize the `default_random_engine` with an `unsigned int` argument that **seeds** the `default_random_engine` to produce a different sequence of random numbers.



Seeding the `default_random_engine`

Figure 5.4 demonstrates seeding the `default_random_engine` (line 15) with an `unsigned int` that you input (line 12). The program produces a different random-number sequence each time it executes, **provided that you enter a different seed**. We used the same seed in the first and third executions, so the same series of 10 numbers is displayed in each. For security, you must ensure that your program seeds the random-number generator differently (and only once) each time the program executes; otherwise, an attacker could determine the sequence of pseudorandom numbers that would be produced.

```
1 // fig05_04.cpp
2 // Randomizing the die-rolling program.
3 #include <iostream>
4 #include <random>
5
```

Fig. 5.4 | Randomizing the die-rolling program. (Part 1 of 2.)

```

6  using namespace std;
7
8 int main() {
9     unsigned int seed{0}; // stores the seed entered by the user
10
11    cout << "Enter seed: ";
12    cin >> seed;
13
14    // set up random-number generation
15    default_random_engine engine{seed}; // seed the engine
16    uniform_int_distribution randomDie{1, 6};
17
18    // display 10 random die rolls
19    for (int counter{1}; counter <= 10; ++counter) {
20        cout << randomDie(engine) << " ";
21    }
22
23    cout << '\n';
24 }
```

Enter seed: 67
6 2 5 6 6 2 2 6 2 1

Enter seed: 432
3 5 6 1 6 1 4 4 2 2

Enter seed: 67
6 2 5 6 6 2 2 6 2 1

Fig. 5.4 | Randomizing the die-rolling program. (Part 2 of 2.)

5.8.4 Seeding the Random-Number Generator with `random_device`

You'll generally seed the random-number generator engine with a `random_device` object (from header `<random>`), as we'll show in Fig. 5.5. A `random_device` produces evenly spread random integers, which cannot be predicted.¹² However, the `random_device` documentation indicates that for performance reasons, it's typically used only to seed random-number generation engines.¹³ Also, `random_device` might be predictable on some platforms.¹⁴ So, check your compiler's documentation before relying on this for secure applications.

Sec 12. “Nondeterministic Algorithm.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Nondeterministic_algorithm.

Perf 13. “`std::random_device`.” Accessed April 14, 2023. https://en.cppreference.com/w/cpp/numeric/random/random_device.

14. “`std::random_device`.” Accessed April 14, 2023. https://en.cppreference.com/w/cpp/numeric/random/random_device.



Checkpoint

1 (*True/False*) When debugging a simulation, you should ensure that each time you run the program, it generates a different sequence of random numbers.

Answer: False. Actually, when debugging a simulation, random-number repeatability is essential for proving that corrections to the program work properly.

2 (*Code*) Rewrite the following distribution used with die rolling so that it produces two `int` values that could be used in a coin-tossing application to represent “heads” or “tails” with equal likelihood. Use a more appropriate `uniform_int_distribution` object name:

```
uniform_int_distribution randomDie{1, 6};
```

Answer: `uniform_int_distribution randomCoin{0, 1};`

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 containing 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 and subsequent rolls. We will define a `rollDice` function to roll the dice and compute and display their sum. The function 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,
- winning on a subsequent roll by “making the point” before rolling a 7,
- losing on the first roll by rolling a 12, and
- losing on a subsequent roll by rolling a 7 before “making the point.”

```
Player rolled 2 + 5 = 7
Player wins
```

```
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 6 + 6 = 12
Player loses
```

```
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 `rollDice`—and the `switch`, `while`, `if...else` and nested `if...else` statements. Function `rollDice`'s prototype (line 8) indicates that the function takes no arguments (empty parentheses) and returns an `int` (the sum of the dice).

```
1 // fig05_05.cpp
2 // Craps simulation.
3 #include <format>
4 #include <iostream>
5 #include <random>
6 using namespace std;
7
8 int rollDice(); // rolls dice, calculates and displays sum
9
```

Fig. 5.5 | Craps simulation.

Scoped enums

The player may win or lose on the first or any subsequent roll. Line 12 declares a user-defined type called a **scoped enumeration** that is introduced by the keywords `enum class`, followed by a type name (`Status`) and a set of identifiers representing integer constants. Line 16 uses the declares variable `gameStatus` as the new type `Status`. This variable can store only values represented by the identifiers in line 12's braces.

```
10 int main() {
11     // scoped enumeration with constants that represent the game status
12     enum class Status {keepRolling, won, lost};
13
14     int myPoint{0}; // point if no win or loss on first roll
15     Status gameStatus{Status::keepRolling}; // game is not over
16
```

The underlying values of these **enumeration constants** are of type `int`. They start at 0 and increment by 1 by default. 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. 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 C++ Core Guidelines state that constants in an `enum class` should use the same naming conventions as variables.¹⁵



To reference a scoped `enum` constant, qualify the constant with the scoped `enum`'s type name (i.e., `Status`) and the scope-resolution operator (`::`), as shown in line 15, 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.

```

17  // determine game status and point (if needed) based on first roll
18  switch (const int sumOfDice{rollDice()}) {
19      case 7: // win with 7 on first roll
20      case 11: // win with 11 on first roll
21          gameStatus = Status::won;
22          break;
23      case 2: // lose with 2 on first roll
24      case 3: // lose with 3 on first roll
25      case 12: // lose with 12 on first roll
26          gameStatus = Status::lost;
27          break;
28      default: // did not win or lose, so remember point
29          myPoint = sumOfDice; // remember the point
30          cout << format("Point is {}\n", myPoint);
31          break; // optional (but recommended) at end of switch
32      }
33

```

The `switch`'s initializer (line 18) creates the variable `sumOfDice` and initializes it by calling `rollDice`. If the roll is 7 or 11, line 21 sets `gameStatus` to `Status::won`. If the roll is 2, 3, or 12, line 26 sets `gameStatus` to `Status::lost`. For other values,

- `gameStatus` remains unchanged (`Status::keepRolling`, which was initially set in line 15),
- line 29 saves `sumOfDice` in `myPoint`, and
- line 30 displays `myPoint`.

Continuing to Roll

After the first roll, if `gameStatus` is `Status::keepRolling`, execution proceeds with the following `while` statement.

15. C++ Core Guidelines. Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps>.

```
34 // while game is not complete
35 while (Status::keepRolling == gameStatus) { // not won or lost
36     // roll dice again and determine game status
37     if (const int sumOfDice{rollDice()}; sumOfDice == myPoint) {
38         gameStatus = Status::won;
39     }
40     else if (sumOfDice == 7) { // Lose by rolling 7 before point
41         gameStatus = Status::lost;
42     }
43 }
44
```

In each loop iteration, the `if` statement's initializer (line 37) calls `rollDice` to produce a new `sumOfDice`. If `sumOfDice` matches `myPoint`, line 38 sets `gameStatus` to `Status::won`, and the loop terminates. If `sumOfDice` is 7, the program sets `gameStatus` to `Status::lost` (line 41), 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`.

```
45 // display won or lost message
46 if (Status::won == gameStatus) {
47     cout << "Player wins\n";
48 }
49 else {
50     cout << "Player loses\n";
51 }
52 }
```

Function `rollDice`

Function `rollDice` rolls two dice (lines 61–62), calculates their sum (line 63), prints their faces and sum (line 66), and returns the sum (line 68).

```
54 // roll dice, calculate sum and display results
55 int rollDice() {
56     // set up random-number generation
57     static random_device rd; // used to seed the default_random_engine
58     static default_random_engine engine{rd()}; // rd() produces a seed
59     static uniform_int_distribution randomDie{1, 6};
60
61     const int die1{randomDie(engine)}; // first die roll
62     const int die2{randomDie(engine)}; // second die roll
63     const int sum{die1 + die2}; // compute sum of die values
64
65     // display results of this roll
66     cout << format("Player rolled {} + {} = {}\n", die1, die2, sum);
67
68     return sum;
69 }
```

Generally, each program that uses random numbers creates one random-number generator engine, seeds it, then uses it throughout the program. In this program, only function `rollDice` needs access to the random-number generator, so lines 57–59 define the objects `random_device`, `default_random_engine` and `uniform_int_distribution` as **static local variables**. Declaring the objects in lines 57–59 **static** ensures they are created only the first time `rollDice` is called. They are then reused in subsequent `rollDice` calls. Unlike other local variables, which exist only until a function call terminates, **static** local retain their values between function calls. The expression `rd()` in line 58 gets a nondeterministic random integer from the `random_device` object and uses it to seed the `default_random_engine`, enabling the program to produce different results each time we execute it.

Additional Notes Regarding Scoped enums

Qualifying an `enum class`'s constant with its type name and `::` explicitly identifies the constant as being in that type's scope. 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, use unique `enum` constant values to help prevent hard-to-find logic errors.

The following scoped enumeration creates the user-defined `enum class` type `Month` with enumeration constants representing the months of the year:

```
enum class Month {jan = 1, feb, mar, apr, may, jun, jul, aug,
    sep, oct, nov, dec};
```

Any constant can be assigned an integer value in the `enum class` definition. Subsequent constants have a value 1 higher than the preceding constant until the next explicit setting. 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. You may assign the same value to multiple constants.

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**¹⁶ allows you to reference an `enum class`'s constants without the type name and scope-resolution operator (`::`). For example, adding the following statement after the `enum class` declaration

```
using enum Status;
```

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 Status::keepRolling;
```

This would allow your code to use `keepRolling` without the `Status::` qualifier. Generally, such `using` declarations should be placed inside the block that uses them.

16. Gašper Ažman and Jonathan Müller, “Using Enum,” July 16, 2019. Accessed April 14, 2023. <http://wg21.link/p1099r5>.



Checkpoint

- 1 (*Fill-in*) Unlike other local variables, which exist only until a function call terminates, a _____ local variable retains its value between function calls.

Answer: static.

- 2 (*Code*) Rewrite the following scoped enumeration so its day-abbreviation-name constants have the underlying values 1 through 7:

```
enum class Day {mon, tues, weds, thurs, fri, sat, sun};
```

Answer: enum class Day {mon = 1, tues, weds, thurs, fri, sat, sun};

5.10 Function-Call Stack and Activation Records

To understand how C++ performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's placed at the *top*—referred to as **pushing** the dish onto the stack. Similarly, when a dish is removed from the pile, it's removed from the top—referred to as **popping** the dish off the stack. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Function-Call Stack

One of the most important mechanisms for computer science students to understand is the **function-call stack** (sometimes referred to as the **program-execution stack**). This data structure—working “behind the scenes”—supports the function call/return mechanism. It also supports the creation, maintenance and destruction of each called function’s local variables. As we’ll see in this section’s diagrams, last-in, first-out (LIFO) behavior is precisely what a function needs to return to the function that called it.

Stack Frames

As each function is called, it may, in turn, call other functions, which may, in turn, call other functions—all before any of the functions return. Each function eventually must return control to the function that called it. So, somehow, the system must keep track of the **return addresses** that each function needs to return control to the function that called it. The function-call stack is the perfect data structure for handling this information. Each time a function calls another function, an entry is **pushed** onto the stack. This entry—called a **stack frame** or an **activation record**—contains the **return address** that the called function needs to return to the calling function. It also contains some additional information we’ll soon discuss. If the called function returns instead of calling another function before returning, the stack frame for the function call is **popped**, and control transfers to the return address in the popped stack frame.

The beauty of the call stack is that each called function always finds the information it needs to return to its caller at the **top** of the call stack. If a function makes calls another function, a stack frame for the new function call is simply **pushed** onto the call stack. Thus, the **return address** required by the newly called function to return to its caller is now at the **top**.

Local Variables and Stack Frames

The stack frames have another important responsibility. Most functions have local variables—parameters and any local variables the function declares. Non-static local variables need to exist while a function is executing. They need to remain active if the function makes calls to other functions. But when a called function returns to its caller, the called function's non-static local variables need to "go away." The called function's stack frame is a perfect place to reserve the memory for the called function's non-static local variables. That stack frame exists as long as the called function is active. When that function returns—and no longer needs its non-static local variables—its stack frame is **popped** from the stack, and those non-static local variables no longer exist.

Stack Overflow

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function-call stack. If more function calls occur than can have their activation records stored on the function-call stack, a fatal error known as **stack overflow** occurs.¹⁷

Function-Call Stack in Action

Now let's consider how the call stack supports the operation of a `square` function called by `main` (lines 9–13 of Fig. 5.6).

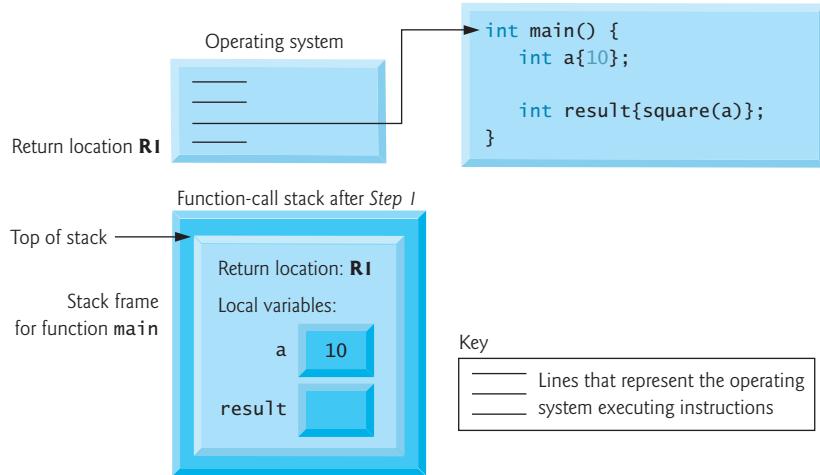
```
1 // fig05_06.cpp
2 // square function used to demonstrate the function
3 // call stack and activation records.
4 #include <iostream>
5 using namespace std;
6
7 int square(int); // prototype for function square
8
9 int main() {
10     int a{10}; // value to square (local variable in main)
11
12     int result{square(a)}; // calculate s squared and store in result
13 }
14
15 // returns the square of an integer
16 int square(int x) { // x is a local variable
17     return x * x; // calculate square and return result
18 }
```

Fig. 5.6 | `square` function used to demonstrate the function-call stack and activation records.

17. This is how the website stackoverflow.com got its name. This is an excellent website for getting answers to your programming questions.

Step 1: Operating System Invokes main to Execute the Application

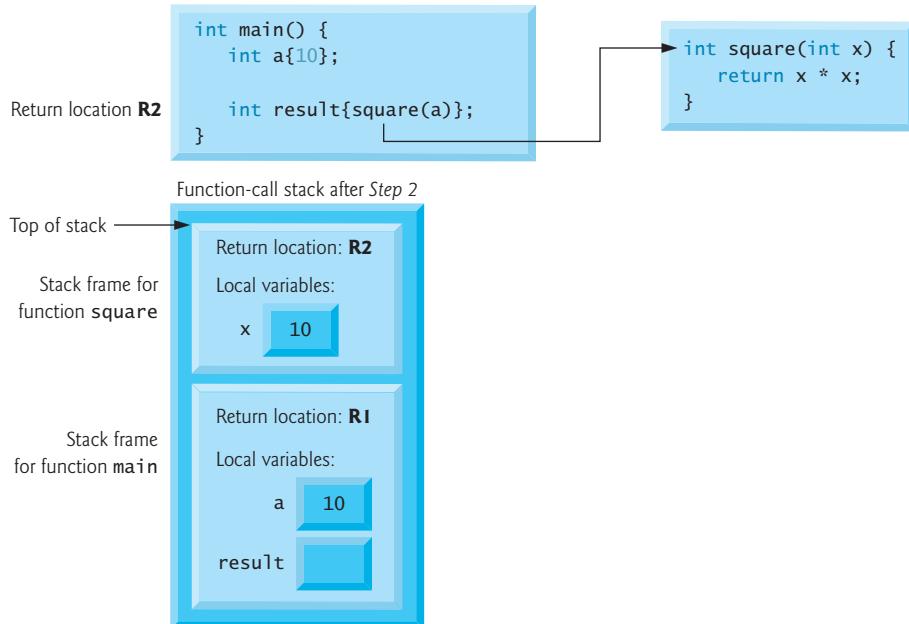
First, the operating system calls `main`—this pushes a stack frame onto the stack, as shown in the following diagram:



The stack frame tells `main` how to return to the operating system (that is, transfer to return address `R1`) and contains the space for `main`'s local variable `a`, which is initialized to 10, and local variable `result`, which is not yet initialized.

Step 2: main Invokes Function square to Perform the Calculation

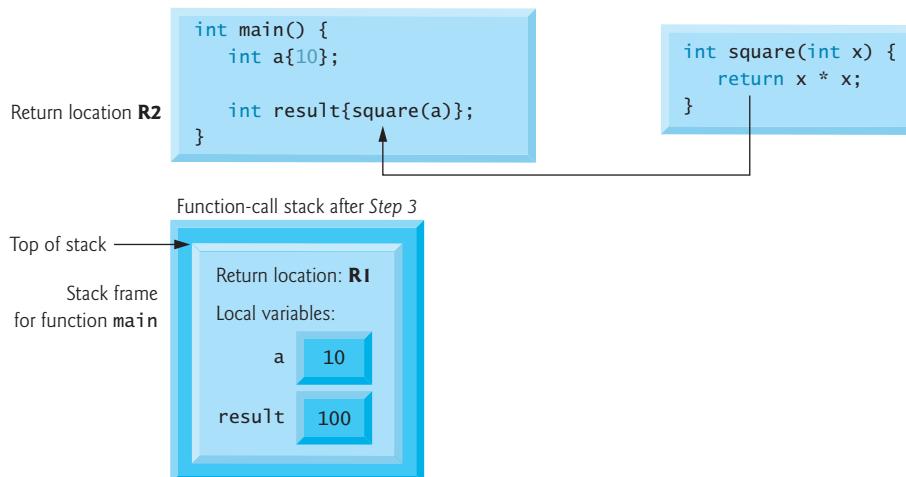
Before returning to the operating system, `main` calls function `square` in line 12 of Fig. 5.6. This pushes a stack frame for `square` (lines 16–18) onto the function-call stack:



This stack frame contains the return address that `square` needs to return to `main` (that is, `R2`) and the memory for `square`'s local variable `x`.

Step 3: `square` Returns Its Result to `main`

After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its variable `x`. So the stack is popped, giving `square` the return location in `main` (that is, `R2`) and losing `square`'s local variable. The following diagram shows the function-call stack after `square`'s stack frame has been popped and `square`'s return value has been stored in `result`:



Reaching the closing right brace of `main` pops its stack frame from the stack. This gives `main` the address it needs to return to the operating system (`R1` in the preceding diagram). At this point, `main`'s local variables `a` and `result` are no longer available.

Data Structures

You've now seen how valuable the stack data structure is in supporting program execution. Data structures have many important applications in computer science. We already introduced C++'s array and vector data structures and continue discussing those and more in Chapter 13, Data Structures: Standard Library Containers and Iterators.

5.11 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 can make 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. Compilers can inline code that you do not explicitly declare **inline**. How-

 ever, the C++ Core Guidelines recommend declaring “small and time-critical” functions `inline`.^{18,19} Figure 5.7 uses the `inline` function `cube` (lines 9–11) to calculate the volume of a cube.

```

1 // fig05_07.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) << '\n';
21 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 5.7 | `inline` function that calculates the volume of a cube.



Checkpoint

- 1 (*True/False*) Placing `inline` before a function’s return type in the function definition forces 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.

Answer: False. Actually, the compiler can ignore the `inline` qualifier.

- 2 (*Fill-in*) Reusable `inline` functions are typically placed in _____ so that their definitions can be inlined in each source file that uses them.

Answer: headers.

- 3 (*True/False*) If you change an `inline` function’s definition, any code that calls that function is automatically recompiled.

Answer: False. Actually, if you change an `inline` function’s definition, you must recompile any code that calls that function.

18. C++ Core Guidelines. Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline>.

19. For an extensive FAQ on the subtleties of inline functions, visit <https://isocpp.org/wiki/faq/inline-functions>.

5.12 References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**. With pass-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 this book has been passed by value. One disadvantage of pass-by-value for large data items is that copying data can take considerable execution time and memory space.



Reference Parameters

This section introduces reference parameters—one of two mechanisms to perform pass-by-reference.^{20,21} 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.



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.



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, place an ampersand (&) after the parameter's type in the function prototype. 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's 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 5.8 compares pass-by-value and pass-by-reference with reference parameters. The argument “styles” in the calls to *squareByValue* and *squareByReference* are identical—both variables are simply mentioned by name. The compiler checks the function prototypes and definitions to determine whether to use pass-by-value or pass-by-reference.

```
1 // fig05_08.cpp
2 // Passing arguments by value and by reference.
3 #include <iostream>
4 using namespace std;
5
```

Fig. 5.8 | Passing arguments by value and by reference. (Part 1 of 2.)

20. Chapter 7 discusses pointers, which enable an alternative form of pass-by-reference.

21. In Chapter 11, we'll discuss another form of reference called an *rvalue* reference.

```

6 int squareByValue(int number); // prototype (for value pass)
7 void squareByReference(int& numberRef); // prototype (for reference pass)
8
9 int main() {
10    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) << '\n';
17    cout << "x = " << x << " after squareByValue\n\n";
18
19    // demonstrate squareByReference
20    cout << "z = " << z << " before squareByReference\n";
21    squareByReference(z);
22    cout << "z = " << z << " after squareByReference\n";
23 }
24
25 // squareByValue multiplies number by itself, stores the
26 // result in number and returns the new value of number
27 int squareByValue(int number) {
28    return number *= number; // caller's argument not modified
29 }
30
31 // squareByReference multiplies numberRef by itself and stores the result
32 // in the variable to which numberRef refers in function main
33 void squareByReference(int& numberRef) {
34    numberRef *= numberRef; // caller's argument modified
35 }
```

```

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 5.8 | Passing arguments by value and by reference. (Part 2 of 2.)

References as Aliases within a Function

References can also be used as aliases for other variables within a function (although they typically are used with functions, as shown in Fig. 5.8). For example, the code

```

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 `count` by using its alias `cRef`. Reference variables must be initialized in their declarations and cannot be reassigned as aliases for 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 *value*—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 that may only appear on the right side of an assignment, such as the result of a calculation.

const References

To specify that a reference parameter should not be allowed to modify the corresponding argument in the caller, place the `const` qualifier before the type name in the parameter's declaration. For example, consider a `displayName` function:

```
void displayName(std::string name) {
    std::cout << name << '\n';
}
```

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
```

Reading this from right to left, the `name` parameter is a reference to a `string` that is 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. 

Returning a Reference to a Local Variable Is Dangerous

When returning a reference to a local non-static variable, the reference refers to a variable that's discarded when the function returns. Attempting to access such a variable yields undefined behavior, often crashing the program or corrupting data.²² References to undefined variables are called **dangling references**. This is a logic error for which compilers often issue a warning. Software-engineering teams often have policies requiring that code must compile without warnings before it can be deployed. You can enforce this for your team by using your compiler's option that causes it to treat warnings as errors.  



Checkpoint

- 1 *(True/False)* Whether you pass a variable pass-by-value or pass-by-reference with reference parameters, the 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.

Answer: True.

- 2 *(True/False)* Reference variables can be reassigned as aliases for other variables.

Answer: False. Reference variables must be initialized in their declarations and cannot be reassigned as aliases for other variables. In this sense, references are constant.

- 3 *(Fill-in)* To specify that a reference parameter should not be allowed to modify the corresponding argument in the caller, place the _____ qualifier before the type name in the parameter's declaration.

Answer: `const`.

22. C++ Core Guidelines. Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-dangle>.

5.13 Default Arguments

It's common for a program to invoke a function from several places with the same argument value for a particular parameter. In such cases, you can specify a **default argument** for the corresponding 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 5.9 demonstrates using default arguments to calculate a box's volume. The function prototype for `boxVolume` (line 7) specifies that all three parameters have default values of 1 by placing = 1 to the right of each parameter.

```

1 // fig05_09.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        << '\n';
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

Fig. 5.9 | Using default arguments. (Part 1 of 2.)

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. 5.9 | Using default arguments. (Part 2 of 2.)

The first call to `boxVolume` (line 11) specifies no arguments, thus using all three default values of 1. The second call (line 15) passes only a `length` argument, thus using default values of 1 for the `width` and `height` arguments. The third call (line 19) passes arguments for only `length` and `width`, thus using a default value of 1 for the `height` argument. The last call (line 23) passes arguments for `length`, `width` and `height`, thus using no default values. Any arguments passed to the function explicitly are assigned to the function's parameters from left to right. Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list). When `boxVolume` receives two arguments, the function assigns those arguments' values to the `length` and `width` parameters in that order. Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to the `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 its prototype, 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.



Checkpoint

- 1** (*True/False*) 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.

Answer: True.

- 2** (*True/False*) 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 its prototype, the default arguments should be specified in the function header.

Answer: True.

- 3** (*True/False*) Default arguments cannot be used with `inline` functions.

Answer: False. Actually, default arguments can be used with `inline` functions.

5.14 Function Overloading

C++ allows functions of the same name to be defined, provided that 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` versions of each math library function introduced in Section 5.3. Overloading functions that perform closely related tasks can make programs clearer.



Overloaded square Functions

Figure 5.10 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 by passing the literal value 7. C++ treats whole-number literal values as `ints`. Similarly, line 21 invokes the `double` version 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.²³ The output confirms that the proper function was called in each case.

```

1 // fig05_10.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 << '\n';
21     cout << square(7.5); // calls double version
22     cout << '\n';
23 }
```

```

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

```

Fig. 5.10 | Overloaded square functions.

23. The C++ standard describes overload resolution at <https://timsong-cpp.github.io/cppwp/n4861/over.match>. For a more detailed explanation, see https://en.cppreference.com/w/cpp/language/overload_resolution.

(Optional) 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 arguments' types conform to the parameters' types. 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 5.11 was compiled with GNU C++. 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++.²⁴

```

1 // fig05_11.cpp
2 // Name mangling to enable type-safe linkage.
3
4 // function square for int values
5 int square(int x) {
6     return x * x;
7 }
8
9 // function square for double values
10 double square(double y) {
11     return y * y;
12 }
13
14 // function that receives arguments of types
15 // int, float, char and int&
16 void nothing1(int a, float b, char c, int& d) { }
17
18 // function that receives arguments of types
19 // char, int, float& and double&
20 int nothing2(char a, int b, float& c, double& d) {
21     return 0;
22 }
23
24 int main() { }
```

```

_Z6squarei
_Z6squared
_Z8nothing1ifcRi
_Z8nothing2ciRfRd
main
```

Fig. 5.11 | 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

24. The command `g++ -S fig05_12.cpp` produces the assembly-language file `fig05_12.s`.

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 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—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

Chapter 11 discusses 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 11.)



Checkpoint

1 *(Fill-in)* _____ is used to create several functions of the same name that perform similar tasks but on data of different types.

Answer: Function overloading.

2 *(True/False)* 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.

Answer: True.

3 (True/False) With operator overloading, you can have a program containing a function that explicitly takes no arguments and a function of the same name that contains all default arguments.

Answer: False. This would cause a compilation error, as the compiler would not know which version of the function to choose to satisfy a call with no arguments.

5.15 Function Templates

Overloaded functions typically perform similar operations on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently with **function templates**. You write a single function template definition, and C++ automatically generates separate **function template specializations** (also called **template instantiations**) to appropriately handle each type of call for the provided argument types. Thus, a single function template defines a family of overloaded functions. Programming with templates is also known as **generic programming**. In this section, we define a custom function template. In subsequent chapters, you'll use many C++ standard library templates, and in Chapter 15, we'll discuss defining custom templates in detail.

maximum Function Template

Figure 5.12 defines a **maximum** function template that returns the largest of three values. All function template definitions begin with the **template keyword** (line 3) followed by a **template parameter list** enclosed in angle brackets (< and >). Every parameter in the template parameter list is preceded by keyword **typename** or keyword **class** (they are synonyms in this context). The **type parameters** are placeholders for fundamental types or user-defined types. These placeholders—in this case, **T**—are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 5). A function template is defined like any other function but uses the type parameters as placeholders for actual data types.

```

1 // Fig. 5.12: maximum.h
2 // Function template maximum header.
3 template <typename T> // or template <class T>
4 T maximum(T value1, T value2, T value3) {
5     T maximumValue{value1}; // assume value1 is maximum
6
7     // determine whether value2 is greater than maximumValue
8     if (value2 > maximumValue) {
9         maximumValue = value2;
10    }
11
12    // determine whether value3 is greater than maximumValue
13    if (value3 > maximumValue) {
14        maximumValue = value3;
15    }
16
17    return maximumValue;
18 }
```

Fig. 5.12 | Function template **maximum** header.

`maximum`'s one type parameter `T` (line 3) is a placeholder for the type of data the function processes. Type parameter names in a template parameter list must be unique. When the compiler encounters a call to `maximum`, it substitutes the argument types in the 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 because we use only one type parameter. We'll use C++ standard library templates that require multiple type parameters in Chapter 13.

Using Function Template `maximum`

Figure 5.13 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). Each call uses arguments of a different type, so “behind the scenes,” the compiler creates a separate function definition for each—one expecting three `int` values, one expecting three `double` values and one expecting three `char` values, respectively.

```

1 // fig05_13.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) << '\n';
34 }
```

Fig. 5.13 | Function template `maximum` test program. (Part I of 2.)

```

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. 5.13 | Function template `maximum` test program. (Part 2 of 2.)

maximum Function Template Specialization for Type int

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

```

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;
}

```



Checkpoint

1 (Fill-In) Overloaded functions typically perform similar operations on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently with _____.

Answer: function templates.

2 (True/False) You write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations (also called template instantiations) to appropriately handle each type of call.

Answer: True.

3 (True/False) Type parameter names in a particular template parameter list must be unique.

Answer: True.

5.16 Recursion

Recursive functions (or methods) call themselves directly or indirectly through other functions (or methods). Recursion can help you solve problems more naturally when an iterative solution is not apparent. We'll show examples and compare the recursive programming style to the iterative style we've used to this point. We'll indicate where each

might be preferable. Recursion is discussed at length in upper-level computer science courses.

5.16.1 Factorials

Consider the factorial of a positive integer n , which is written $n!$ and pronounced “ n factorial.” This 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. You can calculate $5!$ iteratively with a `for` statement, as in:

```
using boost::multiprecision::cpp_int;
cpp_int factorial{1};
int number{5};

for (int counter{number}; counter >= 1; --counter) {
    factorial *= counter;
}
```

5.16.2 Recursive Factorial Example

Recursive problem-solving approaches have several elements in common. When you call a recursive function to solve a problem, it can solve only the **simplest case(s)**, known as the **base case(s)**. If you call the function with a base case, it immediately returns a result. If you call the function with a more complex problem, it typically divides the problem into two pieces—one that the function knows how to do and one that it does not know how to do. To make recursion feasible, this latter piece must be a slightly simpler or smaller version of the original problem. Because this new problem resembles the original problem, the function calls a fresh **copy** of itself to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**. This concept of separating the problem into two smaller portions is known as a **divide-and-conquer** approach.



The recursion step executes while the original function call is still active (i.e., it has not finished executing). It can result in many more recursive calls as the function divides each new subproblem into two conceptual pieces. For the recursion to eventually terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must **converge on a base case**. When the function recognizes the base case, it returns a result to the previous copy of the function. A sequence of returns ensues until the original function call returns the final result to the caller.

Recursive Factorial Approach

You can arrive at a recursive factorial representation by observing that $n!$ can be written as:

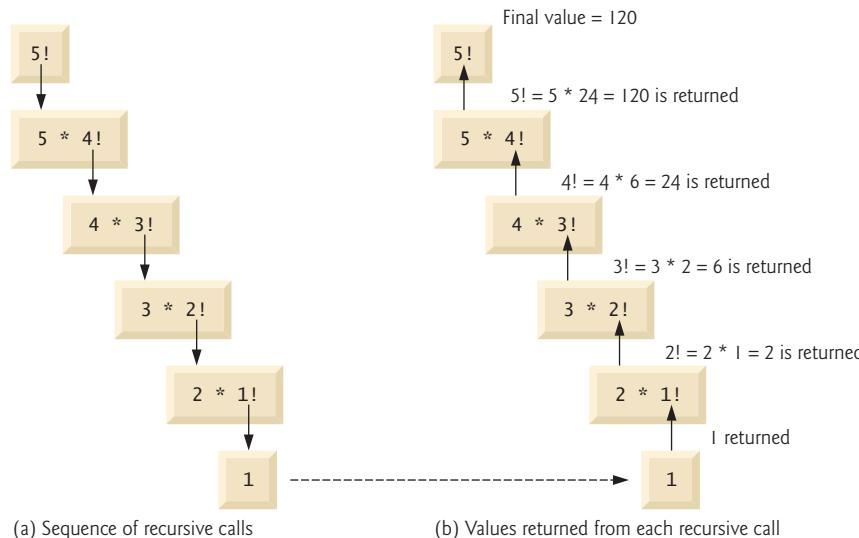
$$n! = n \cdot (n - 1)!$$

For example, $5!$ is equal to $5 \cdot 4!$, as in:

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

Visualizing Recursion

The evaluation of $5!$ would proceed as shown below. The left column shows how the succession of recursive calls proceeds until $1!$ (the base case) is evaluated a 1, terminating the recursion. From bottom to top, the right column shows the values returned from each recursive call to its caller until the final value is calculated and returned.



Implementing a Recursive Factorial Function

Figure 5.14 uses recursion to calculate and print the factorials of the integers 0–10, 20, 30 and 40. The recursive function `factorial` (lines 25–32) first determines whether the base-case terminating condition `number <= 1` (line 26) is true. If so, `factorial` returns 1, and no further recursion is necessary, so the function terminates. If `number` is greater than 1, line 30 expresses the problem as the product of `number` and a recursive call to `factorial` that evaluates `factorial(number - 1)`. This is a slightly smaller problem than the original calculation, `factorial(number)`. Note that function `factorial` must receive a nonnegative argument. We do not test for this case. The loop in lines 13–16 calls the `factorial` function for the values from 0 through 10. Lines 19–21 call the `factorial` function for the values 20, 30 and 40. The output shows that factorial values grow quickly.

```

1 // fig05_14.cpp
2 // Recursive function factorial.
3 #include <boost/multiprecision/cpp_int.hpp>
4 #include <iostream>
5 #include <format>
6 using boost::multiprecision::cpp_int;
7 using namespace std;
8
9 cpp_int factorial(int number); // function prototype
10

```

Fig. 5.14 | Recursive function `factorial`. (Part I of 2.)

```

11 int main() {
12     // calculate the factorials of 0 through 10
13     for (int counter{0}; counter <= 10; ++counter) {
14         cout << format("{:>2}! = ", counter) << factorial(counter)
15         << '\n';
16     }
17
18     // display factorials of 20, 30 and 40
19     cout << format("\n{:>2}! = ", 20) << factorial(20)
20     << format("\n{:>2}! = ", 30) << factorial(30)
21     << format("\n{:>2}! = ", 40) << factorial(40) << '\n';
22 }
23
24 // recursive definition of function factorial
25 cpp_int factorial(int number) {
26     if (number <= 1) { // test for base case
27         return 1; // base cases: 0! = 1 and 1! = 1
28     }
29     else { // recursion step
30         return number * factorial(number - 1);
31     }
32 }
```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

20! = 2432902008176640000
30! = 265252859812191058636308480000000
40! = 815915283247897734345611269596115894272000000000
```

Fig. 5.14 | Recursive function factorial. (Part 2 of 2.)

Factorial Values Grow Quickly

Function factorial receives an `int` and returns a Boost Multiprecision `cpp_int` object (introduced in Section 3.14). The function produces large values so quickly that type `long long` does not help us compute many factorial values before reaching its maximum value. On most systems, a `long long` can store values from $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$ —a maximum of 19 decimal digits (which can represent integers in the quintillions). As you can see from this program’s output, the factorial of 20 is a 19-digit number and the subsequent values displayed have significantly more than 19 digits.

Indirect Recursion

A recursive function may call another function, which may, in turn, make a call back to the recursive function. This is known as an **indirect recursive call** or **indirect recursion**.

For example, function A calls function B, which makes a call back to function A. This is still recursion because the second call to function A is made while the first call to function A is still active. That is, the first call to function A has not yet finished executing (because it is waiting on function B to return a result to it) and has not returned to function A's original caller.

Stack Overflow and Infinite Recursion

The amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function-call stack. If more recursive function calls occur than can have their activation records stored on the stack, a fatal error known as **stack overflow** occurs.²⁵ This typically results from **infinite recursion**, which can be caused by omitting the base case or incorrectly writing the recursion step, so that it does not converge on the base case. Infinite recursion is analogous to an infinite loop in an iterative solution.

Err

Recursion and the Function-Call Stack

In Section 5.10, we introduced the **stack data structure** in the context of understanding how C++ performs function calls. We discussed both the **function-call stack** and **stack frames**. That discussion also applies to recursive function calls. Each recursive function call gets its own stack frame on the function-call stack. When a given call completes, the system pops the function's stack frame from the stack, and control returns to the caller, possibly another copy of the same function.



Checkpoint

1 (True/False) To make recursion feasible, the recursion step in a recursive solution must resemble the original problem but be a slightly larger version of it.

Answer: False. To make recursion feasible, the recursion step in a recursive solution must resemble the original problem but be a slightly **smaller** or **simpler** version of it.

2 (Code) The following code should recursively calculate the sum of the integers from 0 through the argument value `number`. Find the error(s) in the following function and correct the code:

```
int sum(int number) { // assume n is nonnegative
    if (number == 0)
        return 0;
    else
        number + sum(number - 1);
}
```

Answer: The recursion step should return the result of `number + sum(number - 1)`:

```
int sum(int number) { // assume n is nonnegative
    if (number == 0)
        return 0;
    else
        return number + sum(number - 1);
}
```

25. This is how the website stackoverflow.com got its name. This is an excellent site for getting answers to your programming questions.

3 (Code) Rewrite the following iterative factorial calculation code so that the loop counts up rather than down:

```
using boost::multiprecision::cpp_int;
cpp_int factorial{1};

for (int counter{number}; counter >= 1; --counter) {
    factorial *= counter;
}
```

Answer:

```
using boost::multiprecision::cpp_int;
cpp_int factorial{1};

for (int counter{1}; counter <= number; ++counter) {
    factorial *= counter;
}
```

5.16.3 Recursive Fibonacci Series Example

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. This series occurs in nature and describes a form of spiral.

The ratio of successive Fibonacci numbers converges on a constant value of 1.618..., a number known as the **golden ratio** or the **golden mean**. Many people 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 often are designed with a golden-mean length-to-width ratio.

Recursive Fibonacci Definition

The Fibonacci series may be defined recursively as follows:

- fibonacci(0) is defined to be 0
- fibonacci(1) is defined to be 1
- fibonacci(*number*) = fibonacci(*number* – 1) + fibonacci(*number* – 2)

There are *two base cases* for the Fibonacci calculation:

- fibonacci(0) is 0, and
- fibonacci(1) is 1.

Function fibonacci

The program of Fig. 5.15 calculates the *n*th Fibonacci number recursively by using function `fibonacci`. Fibonacci numbers tend to become large quickly, although much more slowly than factorials. Figure 5.15 also shows the execution of the program, which displays the Fibonacci values for several numbers.

```

1 // fig05_15.cpp
2 // Recursive function fibonacci.
3 #include <format>
4 #include <iostream>
5 using namespace std;
6
7 long fibonacci(long number); // function prototype
8
9 int main() {
10    // calculate the fibonacci values of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12        cout << format("fibonacci({}) = {}\n",
13                      counter, fibonacci(counter));
14    }
15
16    // display higher fibonacci values
17    cout << format("\nfibonacci(20) = {}\n", fibonacci(20))
18    << format("fibonacci(30) = {}\n", fibonacci(30))
19    << format("fibonacci(35) = {}\n", fibonacci(35));
20 }
21
22 // recursive function fibonacci
23 long fibonacci(long number) {
24    if ((0 == number) || (1 == number)) { // base cases
25        return number;
26    }
27    else { // recursion step
28        return fibonacci(number - 1) + fibonacci(number - 2);
29    }
30 }
```

```

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. 5.15 | Recursive function fibonacci.

Function `fibonacci` (lines 23–30) recursively calculates the n th Fibonacci number. Each initial call to function `fibonacci` from `main` is *not* a recursive call. However, all subsequent calls to `fibonacci` performed from function `fibonacci`'s block *are* recursive because, at that point, the calls are initiated by the function itself. Each call to `fibonacci` immediately tests for the **base cases**—`number` equal to 0 or `number` equal to 1. If `number`

matches a base case, line 25 returns `number` because `fibonacci(0)` is 0 and `fibonacci(1)` is 1. Interestingly, if `number` is greater than 1, the recursion step generates **two recursive calls**, each for a slightly smaller problem than the original call to `fibonacci`.

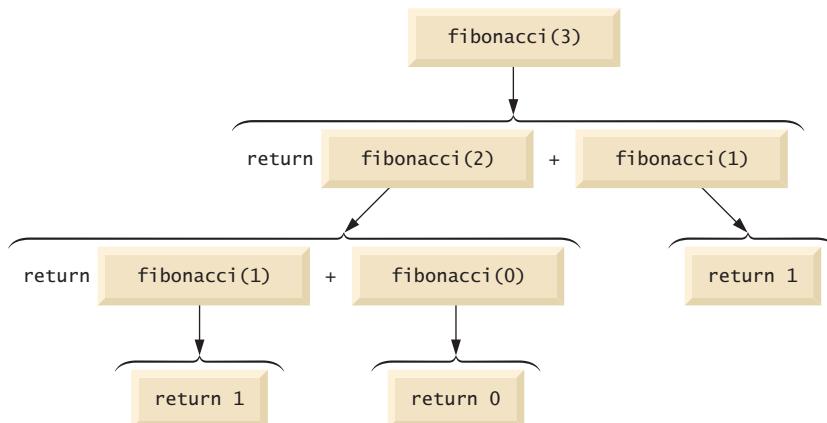
Testing Function `fibonacci`

The `for` loop in lines 11–14 tests `fibonacci`, displaying the Fibonacci values of 0–10. The variable `counter` indicates which Fibonacci number to calculate in each iteration of the loop. Lines 17–19 also calculate Fibonacci values for the numbers 20, 30 and 35. You'll notice that the speed of the calculation slows substantially for higher Fibonacci values.

Analyzing the Calls to Function `fibonacci`

The following diagram shows how function `fibonacci` evaluates `fibonacci(3)`:

- At the diagram's bottom, we're left with the values 1, 0 and 1—the results of evaluating the **base cases**.
- The first two return values (from left to right), 1 and 0, are returned as the values for the calls `fibonacci(1)` and `fibonacci(0)`.
- The sum 1 plus 0 is returned as the value of `fibonacci(2)`. This is added to the result (1) of the rightmost call to `fibonacci(1)`, producing the value 2.
- This final value is then returned as the value of `fibonacci(3)`.



Complexity Issues

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the `fibonacci` function that does not match one of the base cases (0 or 1) results in **two more recursive calls** to the `fibonacci` function. Hence, this set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 with the recursive implementation requires 21,891 calls to the `fibonacci` function; calculating the Fibonacci value of 30 requires 2,692,537 calls!

As you try to calculate larger Fibonacci values, you'll notice that each consecutive Fibonacci number you calculate results in a substantial increase in calculation time and in the number of calls to the `fibonacci` function. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls! As you

can see, the number of calls to `fibonacci` increases quickly—1,664,080 additional calls between Fibonacci values of 30 and 31 and 2,692,538 additional calls between Fibonacci values of 31 and 32! The difference in the number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the difference in the number of calls for Fibonacci values between 30 and 31. Problems of this nature can humble even the world’s most powerful computers.

In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks—that is, how many operations do they perform? Complexity issues are discussed in detail in the upper-level computer science curriculum course, generally called “Algorithms.” We introduce various complexity issues later in Chapter 21, Computer Science Thinking: Searching, Sorting and Big O. In general, you should avoid Fibonacci-style recursive programs that result in an exponential “explosion” of function calls.



Checkpoint

- 1** (*Fill-In*) The ratio of successive Fibonacci numbers converges on a constant value of 1.618..., a number that has been called the _____ or the _____.

Answer: golden ratio, golden mean.

- 2** (*True/False*) In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks.

Answer: True.

- 3** (*Code*) Create a function named `iterative_fibonacci` that uses looping rather than recursion to calculate Fibonacci numbers.

Answer:

```
long iterative_fibonacci(long number) {
    long result{0};
    long temp{1};

    for (int counter{0}; counter < number; ++counter) {
        long previousResult{result};
        result = result + temp;
        temp = previousResult;
    }

    return result;
}
```

5.16.4 Recursion vs. Iteration

We’ve studied functions `factorial` and `fibonacci`, which can be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss 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 (e.g., `for` or `while`), whereas recursion uses a selection statement (e.g., `if` or `if...else` or nested `if...else`):

- Iteration and recursion involve performing tasks repeatedly: Iteration uses an iteration statement, whereas recursion uses repeated function calls.

- Iteration and recursion each involve a **termination test**: Iteration terminates when the loop-continuation condition fails, whereas recursion terminates when a base case is reached.
- Counter-controlled iteration and recursion each **gradually approach termination**: Iteration keeps modifying a counter until the loop-continuation condition fails, whereas recursion keeps producing smaller versions of the original problem until the base case is reached.
- Err ** Iteration and recursion **can occur infinitely**: An infinite loop occurs if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case or if the base case is mistakenly not tested.

Negatives of Recursion

Perf  Recursion has many negatives. It repeatedly invokes the mechanism and, consequently, the **overhead, of function calls**. This overhead can be expensive in terms of processor time and memory space. Each recursive call causes another copy of the function to be created (actually, only the function's variables, stored in the stack frame). This set of copies **can consume considerable memory space**. Iteration avoids these repeated function calls and extra memory assignments. However, iterative solutions are not readily apparent for some algorithms that are easily expressed and understood with recursion.

When to Choose Recursion vs. Iteration

Any problem that can be solved recursively also can be solved iteratively. A recursive approach is usually chosen when it 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.

Summary of Recursion Examples and Exercises in This Book

The following table summarizes the recursion examples and exercises in the text.

Location in text (Part I of 2.)	Recursion examples and exercises
<i>Chapter 5</i>	
Section 5.16.2, Fig. 5.14	Factorial function
Section 5.16.3, Fig. 5.15	Fibonacci function
Exercise 5.24	Recursive exponentiation
Exercise 5.26	Towers of Hanoi
Exercise 5.28	Visualizing recursion
Exercise 5.29	Greatest common divisor
Exercise 5.31, Exercise 5.32	“What does this program do?” exercise

Location in text (Part 2 of 2.)	Recursion examples and exercises
<i>Chapter 6</i>	
Exercise 6.13	“What does this program do?” exercise
Exercise 6.16	“What does this program do?” exercise
Exercise 6.29	Determine whether a string is a palindrome
Exercise 6.30	Eight Queens
Exercise 6.31	Print an array
Exercise 6.32	Print a <code>string</code> backward
Exercise 6.33	Minimum value in an array
<i>Chapter 21</i>	
Section 21.7, Fig. 21.5	Mergesort
Exercise 21.8	Linear search
Exercise 21.9	Binary search
Exercise 21.10	Quicksort



Checkpoint

1 (*Fill-in*) Iteration and recursion each have a termination test. Iteration terminates when the loop-continuation condition fails. Recursion terminates when _____.

Answer: a base case is recognized.

2 (*True/False*) Each recursive call creates another copy of the function’s variables; this can consume considerable memory.

Answer: True.

3 (*True/False*) Recursion is preferred over iteration in performance situations.

Answer: False. If possible, avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

5.17 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 to the end of 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 parameter scope** and are known only in the prototype in which they appear. Later we’ll see other scopes, including **class scope** in Chapter 9 and **function scope** and **namespace scope** in Chapter 20.

Block Scope

Identifiers declared in a block have **block scope**, which begins at the identifier’s declaration and ends at the block’s terminating right brace (`}`). Local variables have block scope, as do

function parameters. Any block can contain variable declarations. In nested blocks, if 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” while the inner block is executing. The inner block “sees” its own local variable’s value and not that of the enclosing block’s identically named variable.



It is typically a logic error to accidentally use 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 visible in the inner block. Avoid variable names in inner scopes that hide names in outer scopes. Most compilers will warn you about this.

As you saw in Fig. 5.5, local variables also may be declared **static**. Such variables also have block scope. 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—it is not reinitialized. The following statement declares a **static** local variable `count` and initializes it to 1:

```
static int count{1};
```

By default, **static** local variables of numeric types are initialized to zero—though explicit initialization is preferred. 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, like `cin` and `cout`, 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 needed to accomplish its designated task, but no more**. An example 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. It also makes your programs easier to read and maintain.

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 5.16 demonstrates scoping issues with global variables, local variables and **static** local variables. We divided this example into smaller pieces with their corresponding out-

puts for discussion purposes. Only the first piece has a figure caption—we'll do this for many subsequent large code examples throughout the book. Line 10 declares and initializes global variable `x` to 1. We'll show that this global variable is hidden in any block (or function) that declares a variable named `x`.

```

1 // fig05_16.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. 5.16 | Scoping example.

Function main

In `main`, line 13 displays the global variable `x`'s value. 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 `main`'s outer block and the global `x`. When the block exits, the `x` with the value 7 is destroyed automatically. Next, line 25 outputs the local variable `x` in the outer block of `main` to show that it's no longer hidden.

```

12 int main() {
13     cout << "global x in main is " << x << '\n';
14
15     const int x{5}; // local variable to main
16
17     cout << "local x in main's outer scope is " << x << '\n';
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 << '\n';
23     }
24
25     cout << "local x in main's outer scope is " << x << '\n';
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` (lines 38–44), `useStaticLocal` (lines 49–57) and `useGlobal` (lines 60–64)—each of which

takes no arguments and returns nothing. The rest of `main` (shown below) calls each function twice in lines 27–32. After calling each function twice, the program prints the local variable `x` in `main` again to show that none of the function calls modified this value because they all referred to variables in other scopes.

```

27  useLocal(); // useLocal has local x
28  useStaticLocal(); // useStaticLocal has static local x
29  useGlobal(); // useGlobal uses global x
30  useLocal(); // useLocal reinitializes its local x
31  useStaticLocal(); // static local x retains its prior value
32  useGlobal(); // global x also retains its prior value
33
34  cout << "\nlocal x in main is " << x << '\n';
35 } // end of main
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 returning program control to its caller. Each time the program calls this function, it recreates local variable `x` and reinitializes it to 25.

```

37 // useLocal reinitializes local variable x during each call
38 void useLocal() {
39     int x{25}; // initialized each time useLocal is called
40
41     cout << "\nlocal x is " << x << " on entering useLocal\n";
42     ++x;
43     cout << "local x is " << x << " on exiting useLocal\n";
44 }
45

```

Function `useStaticLocal`

Function `useStaticLocal` declares static variable `x`, initializing it to 50. Local static variables retain their values, even when they're out of scope (i.e., the enclosing function 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` still contains the value 51. The initialization in line 50 occurs only the first time `useStaticLocal` is called (line 28).

```

46 // useStaticLocal initializes static local variable x only the
47 // first time the function is called; value of x is saved
48 // between calls to this function
49 void useStaticLocal() {
50     static int x{50}; // initialized first time useStaticLocal is called
51
52     cout << "\nlocal static x is " << x
53         << " on entering useStaticLocal\n";
54     ++x;
55     cout << "local static x is " << x
56         << " on exiting useStaticLocal\n";
57 }
58

```

Function `useGlobal`

Function `useGlobal` does not declare any variables. So, 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 still has its modified value, 10.

```

59 // useGlobal modifies global variable x during each call
60 void useGlobal() {
61     cout << "\nglobal x is " << x << " on entering useGlobal\n";
62     x *= 10;
63     cout << "global x is " << x << " on exiting useGlobal\n";
64 }

```



Checkpoint

1 (*True/False*) In nested blocks, if an identifier in an outer block has the same name as an identifier in an inner block, the one in the inner block is “hidden” until the outer block terminates.

Answer: False. Actually, in nested blocks, if 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.

2 (*True/False*) In general, variables should be declared in the widest scope in which they need to be accessed.

Answer: False. Actually, variables should be declared in the narrowest scope in which they need to be accessed.

3 (Code) For the following program, state the scope (global namespace scope or block scope) of each of the following elements:

- The variable `x` in `main`.
- The variable `y` in function `cube`'s definition.
- The function `cube`.
- The function `main`.
- The function prototype for `cube`.

```

1 #include <iostream>
2 using namespace std;
3
4 int cube(int y); // function prototype
5
6 int main() {
7     int x{0};
8
9     for (x = 1; x <= 10; x++) { // loop 10 times
10        cout << cube(x) << '\n'; // calculate cube of x and output results
11    }
12 }
13
14 // definition of function cube
15 int cube(int y) {
16     return y * y * y;
17 }
```

Answer: a) block scope. b) block scope. c) global namespace scope. d) global namespace scope. e) global namespace scope.

5.18 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 5.17 shows the unary scope resolution operator with global and local 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`. In general, you should 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. Generally, global variables are discouraged. If you use one, always use the unary scope resolution operator (`::`) to refer to it (even if there is no collision with a local-variable name). This makes it clear that you're accessing a global 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.

```
1 // fig05_17.cpp
2 // Unary scope resolution operator.
3 #include <iostream>
4 using namespace std;
5
6 const int number{7}; // global variable named number
7
8 int main() {
9     const 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 << '\n';
14 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

Fig. 5.17 | Unary scope resolution operator.



Checkpoint

- I (*True/False*) The unary scope resolution operator can be used to access a local variable of the same name in an outer block.

Answer: False. Actually, the unary scope resolution operator cannot access local variables of the same name in outer blocks. It can be used only to access a global variable when a local variable of the same name is in scope.

5.19 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

You probably 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 *Vigenère secret key cipher*.²⁶

In previous 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 `Cipher` class (in the header "`cipher.h`") and provided it to you.

For this Objects Natural section, we show you how easy it is to do encryption and decryption with an object of a pre-existing class. In Chapter 9, Custom Classes, you'll build custom classes. Section 9.22 explains the Vigenère secret key cipher algorithm and gives you the opportunity to study our custom class to see how it implements the cipher.

26. “Vigenère Cipher,” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Vigenère_cipher.

Cryptography

Cryptography has been used for thousands of years^{27,28} 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, as we do at deitel.com.



Caesar Cipher

Julius Caesar used a simple **substitution cipher** to encrypt military communications.²⁹ His technique—known as the **Caesar cipher**³⁰—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 plaintext

Caesar Cipher

would be encrypted as

Fdhvdu Flskhu

The encrypted text is known as the **ciphertext**.

Vigenère 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 encrypted with the Caesar cipher and assume with a high likelihood that the character appearing most frequently is probably an “e.”

In this example, you'll use a Vigenère cipher, which is a secret-key substitution cipher. This cipher is implemented using 26 Caesar ciphers—one for each letter of the alphabet. A Vigenère cipher uses letters from the plaintext and secret key to look up replacement characters in the various Caesar ciphers.³¹

For this cipher, the secret key must be composed of letters. Like passwords, the secret key should not be easy to guess. 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 originally used to create the ciphertext. Presumably, you'd provide that in advance—possibly in a face-to-face meeting.

Using Our Cipher Class

For the example in Fig. 5.18, you'll use our class **Cipher**, which implements the Vigenère cipher. The header "cipher.h" (line 3) from the ch05 examples folder defines the class.

-
- 27. “Cryptography.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis.
 - 28. Binance Academy. “History of Cryptography.” Updated December 23, 2022. Accessed April 14, 2023. <https://www.binance.vision/security/history-of-cryptography>.
 - 29. “Caesar Cipher.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Caesar_cipher.
 - 30. 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>.
 - 31. See https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher to learn about the implementation.

You don't need to read and understand the class's code to use its encryption and decryption capabilities. You simply create an object of class `Cipher` then call its `encrypt` and `decrypt` member functions to encrypt and decrypt text, respectively. Lines 11 and 15 call the `getline` function call to read all the characters you type until you press *Enter*. Here, the characters come from `cin` and are placed into the `string` in the second argument.

```

1 // fig05_18.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 << '\n';
22
23    // decrypt cipherText
24    cout << "\nDecrypted:\n"   "
25        << cipher.decrypt(cipherText, secretKey) << '\n';
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) << '\n';
32 }
```

```

Enter the text to encrypt:
Welcome to Modern C++ application development with C++20!
Enter the secret key:
XMWUJBVYHXZ
Encrypted:
Tqhwxnz rv Jnaqn h L++ bknsfbxf eiw eztlinmyahc xdro Z++20!
Decrypted:
Welcome to Modern C++ application development with C++20!
```

Fig. 5.18 | Encrypting and decrypting text with a Vigenère cipher. (Part I of 2.)

```
Enter the ciphertext to decipher:  
Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwqhlz
```

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

Fig. 5.18 | Encrypting and decrypting text with a Vigenère cipher. (Part 2 of 2.)

Class Cipher's Member Functions

The class provides two key member functions:

- `encrypt` receives `strings` representing the plaintext to encrypt and the secret key, uses the Vigenère cipher to encrypt the text, then returns a `string` containing the ciphertext.
- `decrypt` receives `strings` representing the ciphertext to decrypt and the secret key, reverses the Vigenère cipher to decrypt the text, then returns a `string` containing the plaintext.

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 plaintext 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

Lines 28–29 prompt for and input existing ciphertext, then lines 30–31 decrypt the ciphertext and display the original plaintext we encrypted.



Checkpoint

1 *(Fill-in)* Most websites now use the _____ protocol to encrypt and decrypt your web interactions.

Answer: HTTPS.

2 *(Fill-in)* The Caesar cipher is an example of a simple _____ cipher.

Answer: substitution.

5.20 Wrap-Up

In this chapter, we presented function features, 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 generate nondeterministic random numbers. 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. Finally, the Objects-Natural case study explored 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.

Exercises

5.1 Show the value of *x* after each of the following statements is performed:

- a) `x = fabs(7.5);`
- b) `x = floor(7.5);`
- c) `x = fabs(0.0);`
- d) `x = ceil(0.0);`
- e) `x = fabs(-6.4);`
- f) `x = ceil(-6.4);`
- g) `x = ceil(-fabs(-8 + floor(-5.5)));`

5.2 (*Parking Charges*) A parking garage charges a \$20.00 minimum fee to park for up to three hours. The garage charges an additional \$5.00 per hour for each hour or part thereof over three hours. The maximum charge for any given 24-hour period is \$50.00. Assume that no car parks for longer than 24 hours at a time. Write a program that calculates and prints the parking charges for each of three customers who parked their cars in this garage yesterday. You should enter the hours parked for each customer. Your program should print the results in a neat tabular format and calculate and print the total of yesterday's receipts. The program should use the function `calculateCharges` to determine the charge for each customer. Your outputs should appear in the following format:

Car	Hours	Charge
1	1.5	20.00
2	4.0	25.00
3	24.0	50.00
TOTAL	29.5	95.50

5.3 (*Rounding Numbers*) An application of function `floor` is rounding a value to the nearest integer. The statement

```
y = floor(x + 0.5);
```

rounds the number *x* to the nearest integer and assigns the result to *y*. Write a program that reads several numbers and uses the preceding statement to round each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

5.4 (Rounding Numbers) Function `floor` can be used to round a number to a specific decimal place. The statement

```
y = floor(x * 10 + 0.5) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = floor(x * 100 + 0.5) / 100;
```

rounds `x` to the hundredths position (the second position to the right of the decimal point). Write a program that defines four functions to round a number `x` in various ways:

- `roundToInteger(number)`
- `roundToTenths(number)`
- `roundToHundredths(number)`
- `roundToThousandths(number)`

For each value read, your program should print the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

5.5 (Short-Answer Questions) Answer each of the following questions:

- What does it mean to choose numbers “at random?”
- Why is the `rand` function useful for simulating games of chance?
- Why would you randomize a program? Under what circumstances is it desirable not to randomize?
- Why is computerized simulation of real-world situations a useful technique?

5.6 (Random Numbers) Write statements that assign random integers to the variable `n` in the following ranges:

- $1 \leq n \leq 2$
- $1 \leq n \leq 100$
- $0 \leq n \leq 9$
- $1000 \leq n \leq 1112$
- $-1 \leq n \leq 1$
- $-3 \leq n \leq 11$

5.7 (Random Numbers) Write statements that print a number at random from each of the following sets:

- 2, 4, 6, 8, 10.
- 3, 5, 7, 9, 11.
- 6, 10, 14, 18, 22.

5.8 (Exponentiation) Write a function `integerPower(base, exponent)` that returns the value of

$$\text{base}^{\text{exponent}}$$

For example, `integerPower(3, 4) = 3 * 3 * 3 * 3`. Assume that `exponent` is a positive, nonzero integer and that `base` is an integer. Do not use any math library functions.

5.9 (Hypotenuse Calculations) Define a function `hypotenuse` that calculates the hypotenuse of a right triangle when the other two sides are given. The function should take two `double` arguments and return the hypotenuse as a `double`. Use this function in a program to determine the hypotenuse for each triangle shown below.

Triangle	Side 1	Side 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

5.10 (*Multiples*) Write a function `isMultiple` that determines for a pair of integers whether the second is a multiple of the first. The function should take two integer arguments and return `true` if the second is a multiple of the first and `false` otherwise. Use this function in a program that inputs a series of pairs of integers.

5.11 (*Even Numbers*) Write a program that inputs a series of integers and passes them one at a time to function `isEven`, which uses the remainder operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise.

5.12 (*Separating Digits*) Write program segments that accomplish each of the following:

- Calculate the integer part of the quotient when integer `a` is divided by integer `b`.
- Calculate the integer remainder when integer `a` is divided by integer `b`.
- Use the program pieces developed in (a) and (b) to write a function that inputs an integer between 1 and 32767 and prints it as a series of digits, each pair separated by two spaces. For example, the integer 4562 should print as follows:

```
4 5 6 2
```

5.13 (*Calculating Number of Seconds*) Write a function that takes the time as three integer arguments (hours, minutes and seconds) and returns the number of seconds since the last time the clock “struck 12.” Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

5.14 (*Celsius and Fahrenheit Temperatures*) Implement the following integer functions:

- Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature.
- Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature.
- Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

5.15 (*Find the Minimum*) Write a program that inputs three double-precision, floating-point numbers and passes them to a function that returns the smallest number.

5.16 (*Perfect Numbers*) An integer is said to be a *perfect number* if the sum of its divisors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number because $6 = 1 + 2 + 3$. Write a function `isPerfect` that determines whether the parameter `number` is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the divisors of each perfect number to confirm that the number is indeed perfect.

5.17 (Prime Numbers) An integer is said to be **prime** if it's divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

- Write a function that determines whether a number is prime.
- Use this function in a program that determines and prints all the prime numbers between 2 and 10,000. How many of these numbers do you have to test before being sure you've found all the primes?
- Initially, you might think that $n/2$ is the upper limit for which you must test to see whether a number is prime, but you need only go as high as the square root of n . Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

5.18 (Reverse Digits) Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

5.19 (Greatest Common Divisor) The **greatest common divisor (GCD)** of two integers is the largest integer that evenly divides each number. Write a function `gcd` that returns the greatest common divisor of two integers.

5.20 (Quality Points for Numeric Grades) Write a function `qualityPoints` that inputs a student's average and returns 4 if a student's average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69 and 0 if the average is lower than 60.

5.21 (Coin Tossing) Write a program that simulates coin tossing. For each coin toss, the program should print `Heads` or `Tails`. Let the program toss the coin 100 times and count the number of times each side of the coin appears. Print the results. The program should call a separate function `flip` that takes no arguments and returns 0 for tails and 1 for heads. [Note: If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time.]

5.22 (Guess-the-Number Game) Write a program that plays the game of “guess the number” as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then displays the following:

```
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.
```

The player then types a first guess. The program responds with one of the following:

- Excellent! You guessed the number!
Would you like to play again (y or n)?
- Too low. Try again.
- Too high. Try again.

If the player's guess is incorrect, your program should loop until the player finally gets the number right. Your program should keep telling the player `Too high` or `Too low` to help the player “zero in” on the correct answer.

5.23 (Guess-the-Number Game Modification) Modify the program of Exercise 5.22 to count the number of guesses the player makes. If the number is 10 or fewer, print "Either

you know the secret or you got lucky!" If the player guesses the number in 10 tries, print "Ahah! You know the secret!" If the player makes more than 10 guesses, print "You should be able to do better!" Why should it take no more than 10 guesses? Well, with each "good guess," the player should be able to eliminate half of the numbers. Now show why any number from 1 to 1000 can be guessed in 10 or fewer tries.

5.24 (Recursive Exponentiation) Write a recursive function `power(base, exponent)` that, when invoked, returns

$$\text{base}^{\text{exponent}}$$

For example, $\text{power}(3, 4) = 3 * 3 * 3 * 3$. Assume that `exponent` is an integer greater than or equal to 1. Hint: The recursion step would use the relationship

$$\text{base}^{\text{exponent}} = \text{base} * \text{base}^{\text{exponent} - 1}$$

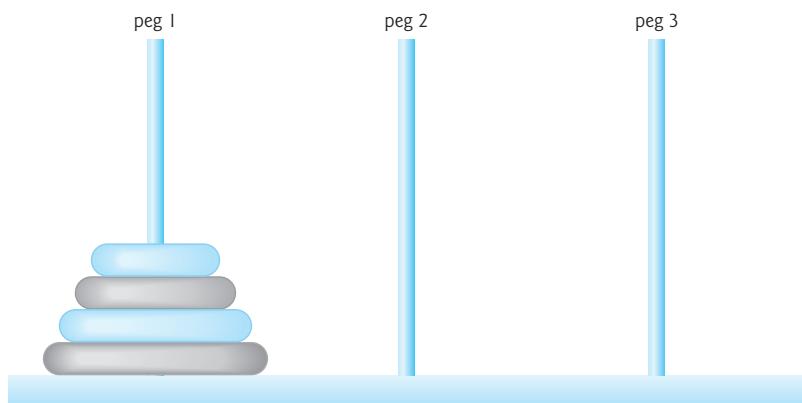
and the terminating condition occurs when the `exponent` is equal to 1 because

$$\text{base}^1 = \text{base}$$

5.25 (Fibonacci Series: Iterative Solution) Write a nonrecursive version of the function `fibonacci` from Fig. 5.15.

5.26 (Towers of Hanoi) In this chapter, you studied functions that can be easily implemented recursively and iteratively. In this exercise, we present a problem whose recursive solution demonstrates the elegance of recursion and whose iterative solution may be less apparent.

The **Towers of Hanoi** is one of the most famous classic problems every budding computer scientist must grapple with. Legend has it that in a temple in the Asia, priests are attempting to move a stack of golden disks from one diamond peg to another. The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from one peg to another under the constraints that exactly one disk is moved at a time, and at no time may a larger disk be placed above a smaller disk. Three pegs are provided, one being used for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts. The following diagram shows the Towers of Hanoi for the case with four disks:



Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that prints the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, attacking this problem with recursion in mind allows the steps to be simple. Moving n disks can be viewed in terms of moving only $n - 1$ disks (hence, the recursion), as follows:

- Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
- Move the last disk (the largest) from peg 1 to peg 3.
- Move the $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ disk (i.e., the base case). This task is accomplished by simply moving the disk without the need for a temporary holding area. Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

- The number of disks to be moved
- The peg on which these disks are initially threaded
- The peg to which this stack of disks is to be moved
- The peg to be used as a temporary holding area

Display the precise instructions for moving the disks from the starting peg to the destination peg. To move a stack of three disks from peg 1 to peg 3, the program displays the following moves:

```
1 → 3 (This means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3
```

5.27 (Towers of Hanoi: Iterative Version) Any program that can be implemented recursively can be implemented iteratively, although sometimes with more difficulty and less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version developed in Exercise 5.26. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

5.28 (Visualizing Recursion) It's interesting to watch recursion "in action." Modify the factorial function of Fig. 5.14 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

5.29 (Recursive Greatest Common Divisor) The greatest common divisor of integers x and y is the largest integer that evenly divides both x and y . Write a recursive function `gcd` that returns the greatest common divisor of x and y , defined recursively as follows: If y is

equal to 0, then $\text{gcd}(x, y)$ is x ; otherwise, $\text{gcd}(x, y)$ is $\text{gcd}(y, x \% y)$, where $\%$ is the remainder operator. [Note: For this algorithm, x must be larger than y .]

5.30 (Distance Between Points) Write function `distance` that calculates the distance between two points (x_1, y_1) and (x_2, y_2) . All numbers and return values should be of type `double`.

5.31 What does the following program do?

```

1 // ex05_31.cpp
2 // What does this program do?
3 #include <iostream>
4 using namespace std;
5
6 int mystery(int, int); // function prototype
7
8 int main() {
9     cout << "Enter two integers: ";
10    int x{0};
11    int y{0};
12    cin >> x >> y;
13    cout << "The result is " << mystery(x, y) << endl;
14 }
15
16 // Parameter b must be a positive integer to prevent infinite recursion
17 int mystery(int a, int b) {
18     if (1 == b) { // base case
19         return a;
20     }
21     else { // recursion step
22         return a + mystery(a, b - 1);
23     }
24 }
```

5.32 After you determine what the program of Exercise 5.31 does, modify the program to function properly after removing the restriction that the second argument be nonnegative.

5.33 (Math Library Functions) Write a program that tests the math library functions in Section 5.3's table. Exercise each of these functions by having your program print out tables of return values for various argument values.

5.34 (Find the Error) Find the error in each of the following program segments and explain how to correct it:

- a) `float cube(float); // function prototype`

- `cube(float number) { // function definition`
- `return number * number * number;`
- `}`
- b) `double square(double number) {`
- `double number{0};`
- `return number * number;`
- `}`

```
c) int sum(int n) {
    if (0 == n) {
        return 0;
    }
    else {
        return n + sum(n);
    }
}
```

5.35 (Craps Game Modification) Modify the craps program of Fig. 5.5 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable bankBalance to 1000. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to bankBalance and, if not, prompt the user to re-enter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check on whether bankBalance has become zero and, if so, print the message "Sorry. You busted!" As the game progresses, print various messages to create some "chatter" such as "Oh, you're going for broke, huh?", "Aw cmon, take a chance!" or "You're up big. Now's the time to cash in your chips!".

5.36 (Circle Area) Write a C++ program that prompts the user for the radius of a circle, then calls inline function circleArea to calculate the area of that circle.

5.37 (Pass-by-Value vs. Pass-by-Reference) Write a complete C++ program with the two alternate functions specified below, each of which simply triples the variable count defined in main. Then compare and contrast the two approaches. These two functions are

- a) function tripleByValue that passes a copy of count by value, triples the copy and returns the new value and
- b) function tripleByReference that passes count by reference via a reference parameter and triples the original value of count through its alias (i.e., the reference parameter).

5.38 (Unary Scope Resolution Operator) What's the purpose of the unary scope resolution operator?

5.39 (Function Template minimum) Write a program that uses a function template called minimum to determine the smaller of two arguments. Test the program using integer, character and floating-point number arguments.

5.40 (Function Template maximum) Write a program that uses a function template called maximum to determine the larger of two arguments. Test the program using integer, character and floating-point number arguments.

5.41 (Find the Error) Determine whether the following program segments contain errors. For each error, explain how it can be corrected. [Note: For a particular program segment, it's possible that no errors are present.]

```
a) template <typename A>
int sum(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}
```

- b) template <A>
A product(A num1, A num2, A num3) {
 return num1 * num2 * num3;
}
c) double cube(int);
int cube(int);

5.42 (*Scoped enum*) Create a scoped enum named `AccountType` containing constants named `savings`, `checking` and `investment`.

5.43 (*Function Prototypes and Definitions*) Explain the difference between a function prototype and a function definition.

Computer Assisted Instruction

Computers create exciting possibilities for improving the educational experience of all students worldwide, as suggested by the next five exercises.

5.44 (*Computer-Assisted Instruction*) The use of computers in education is referred to as *computer-assisted instruction (CAI)*. Write a program that will help an elementary-school student learn multiplication. Use the `rand` function to produce two positive one-digit integers. The program should then prompt the user with a question, such as

How much is 6 times 7?

The student then inputs the answer. Next, the program checks the student's answer. If correct, display the message "Very good!" and ask another multiplication question. If the answer is wrong, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the application begins execution and each time the user answers the question correctly.

5.45 (*Computer-Assisted Instruction: Reducing Student Fatigue*) One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise 5.44 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

Very good!
Excellent!
Nice work!
Keep up the good work!

Possible responses to an incorrect answer:

No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a `switch` statement to issue the responses.

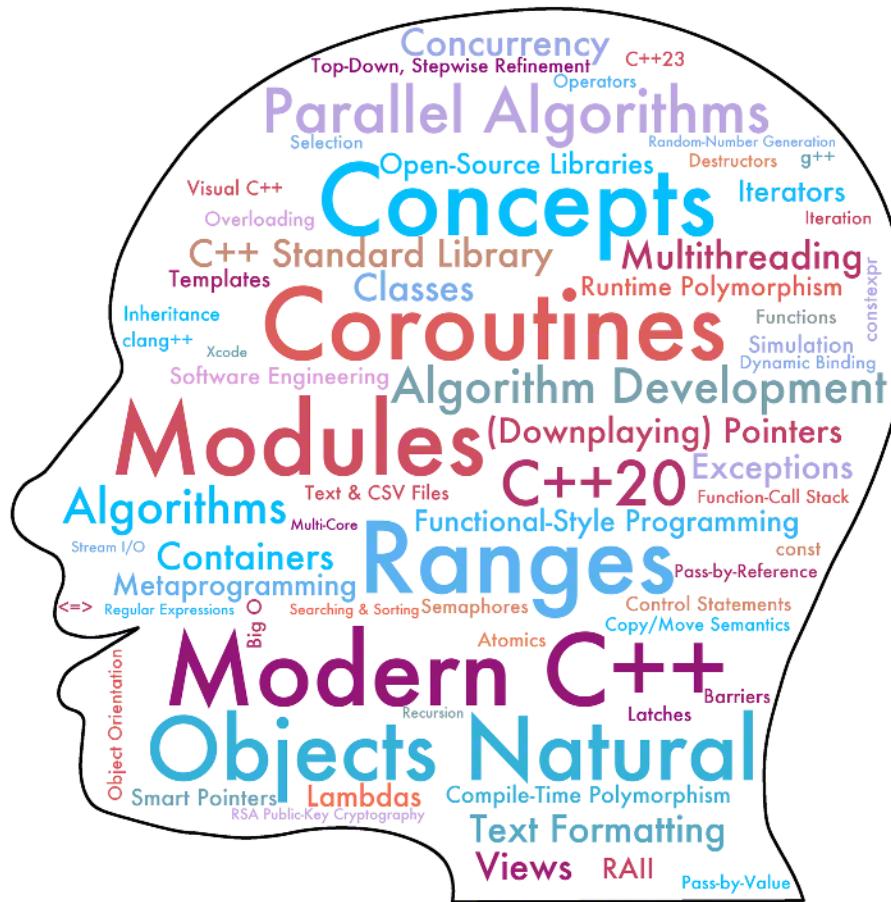
5.46 (*Computer-Assisted Instruction: Monitoring Student Performance*) More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 5.45 to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage that are correct. If the percentage is below 75%, display "Please ask your teacher for extra help.", then reset the program so another student can try it. If the percentage is 75% or higher, display "Congratulations, you are ready to go to the next level!", then reset the program so another student can try it.

5.47 (*Computer-Assisted Instruction: Difficulty Levels*) Exercises 5.44–5.46 developed a computer-assisted instruction program to help teach an elementary-school student multiplication. Modify the program to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems; at a difficulty level of 2, numbers as large as two digits, and so on.

5.48 (*Computer-Assisted Instruction: Varying the Types of Problems*) Modify the program of Exercise 5.47 to allow the user to pick a type of arithmetic problem to study. Option 1 means addition problems, 2 means subtraction problems, 3 means multiplication problems, 4 means division problems, and 5 means a random mixture of all these types.

6

arrays, vectors, 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.
- Perform common array manipulations.
- Visualize an array's data.
- Sort array elements in ascending order.
- Quickly determine whether a sorted array contains a specific value 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	6.11 Using arrays to Summarize Survey Results
6.2 arrays	6.12 Sorting and Searching arrays
6.3 Declaring arrays	6.13 Multidimensional arrays
6.4 Initializing array Elements in a Loop	6.14 Intro to Functional-Style Programming
6.5 Initializing an array with an Initializer List	6.14.1 What vs. How
6.6 Range-Based for Statement	6.14.2 Passing Functions as Arguments to Other Functions: Introducing Lambda Expressions
6.7 Calculating array Element Values and an Intro to <code>constexpr</code>	6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library
6.8 Totaling array Elements	6.15 Objects-Natural Case Study: C++ Standard Library Class Template <code>vector</code>
6.9 Using a Primitive Bar Chart to Display array Data Graphically	6.16 Wrap-Up
6.10 Using array Elements as Counters	

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:

- fixed-size **arrays**¹ 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 order. We show that attempting to access data outside an array's or vector's bounds might cause an exception—a runtime indication that a problem occurred. Then we use exception handling to resolve (or handle) that exception. Chapter 12 covers exceptions in more depth.

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 improve 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.

Fully Qualified Standard Library Names

Since Section 2.7, we’ve included “`using namespace std;`” in each program, so you did not need to qualify every C++ standard library feature with “`std:::`”. In larger systems, `using` directives can lead to subtle, difficult-to-find bugs. Going forward, we’ll fully qualify most identifiers from the C++ standard library.

1. In Chapter 7, you’ll see that C++ also provides a language element called an array (different from the `array` container). Modern C++ code should use class template `array` rather than traditional arrays.





Checkpoint

1 (Fill-in) A(n) _____ is a runtime indication that a problem occurred.

Answer: exception.

2 (Fill-in) Two C++ array-like collections are arrays and _____.

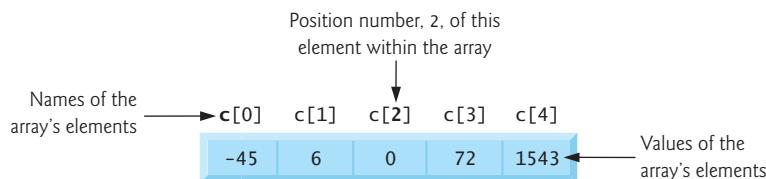
Answer: vectors.

3 (True/False) In larger systems, using directives can lead to subtle, difficult-to-find bugs, so it is recommended that you fully qualify most identifiers from the C++ standard library.

Answer: True.

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 expression. The brackets enclosing an index are an operator with the second highest level of precedence. The result of that operator is an *lvalue*—it can be used on the left side of an assignment, just as a variable name can. For example, the following statement replaces `c[4]`'s value:

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

See https://en.cppreference.com/w/cpp/language/operator_precedence for the complete operator precedence chart.



Checkpoint

1 (Fill-in) The elements of 5-element array `c` are named _____.

Answer: `c[0]` through `c[4]`.

2 (True/False) The highest index in an array is always the same as the array's number of elements.

Answer: False. Actually, the highest index in an array is always one less than the array's number of elements.

- 3 (Code) Write a statement that places the sum of array elements `c[0]` and `c[1]` into array element `c[2]`.

Answer: `c[2] = c[0] + c[1];`

6.3 Declaring arrays

To specify an array's element type and number of elements, use a declaration of the form

```
std::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 **class template specializations** for various types—such as an array of five `ints`, an array of seven `doubles` or an array of 100 `Employees`. You'll begin creating custom types in Chapter 9. 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 this declaration:

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



Checkpoint

- 1 (Fill-in) The notation

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

indicates that `array` is a class _____.

Answer: template.

- 2 (Fill-In) Like function templates, the compiler can use class templates to create _____ for various types, such as an array of five `ints`, an array of seven `doubles` or an array of 100 `Employees`.

Answer: class-template specializations.

6.4 Initializing array Elements in a Loop

The program in Fig. 6.1 declares five-element integer array `values` (line 8). Line 5 includes the `<array>` header containing the class template `array`'s definition.

```

1 // fig06_01.cpp
2 // Initializing an array
3 #include <iostream>
4 #include <array>
5
6
7 int main() {
8     std::array<int, 5> values; // values is an array of 5 int values

```

Fig. 6.1 | Initializing an array's elements to zeros and printing the array. (Part I of 2.)

```
9      // initialize elements of array values to 0
10     for (size_t i{0}; i < values.size(); ++i) {
11         values[i] = 0; // set element at location i to 0
12     }
13
14
15     std::cout << std::format("{:>7}{:>10}\n", "Element", "Value");
16
17     // output each array element
18     for (size_t i{0}; i < values.size(); ++i) {
19         std::cout << std::format("{:>7}{:>10}\n", i, values[i]);
20     }
21
22     std::cout << std::format("\n{:>7}{:>10}\n", "Element", "Value");
23
24     // access elements via the at member function
25     for (size_t i{0}; i < values.size(); ++i) {
26         std::cout << std::format("{:>7}{:>10}\n", i, values.at(i));
27     }
28
29     // accessing an element outside the array
30     values.at(10); // throws an exception
31 }
```

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 (which is 5)
Aborted
```

Fig. 6.1 | Initializing an array's elements to zeros and printing the array. (Part 2 of 2.)

Assigning Values to array Elements in a Loop

Lines 11–13 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 but elements of static arrays are. In a loop that processes array elements, ensure that the loop's condition prevents accessing elements outside the array's bounds. Section 6.6 presents the range-based `for` statement, which provides a safer way to process every element of an array.



Type `size_t`

Lines 11, 18 and 25 declare each loop's control variable as type `size_t`. The C++ standard specifies that `size_t` is an unsigned integral type that can represent the size of any object.²

Use this type for any variable representing an array's size or indices. Type `size_t` is defined in the `std` namespace and is in header `<cstddef>`, which is typically included for you by other standard library headers that use `size_t`. If you compile a program and receive errors indicating `size_t` is not defined, add `#include <cstddef>` to the program.

Displaying the array Elements

The first output statement (line 15) displays the column headings for the rows printed in the subsequent `for` statement (lines 18–20). These output statements use the C++20 text-formatting features introduced in Section 4.6 to output the array in tabular format.



Avoiding a Security Vulnerability: Bounds Checking for array Indices

When you use the `[]` operator to access an array element (as in lines 11–13 and 18–20), 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**, however, performs bounds checking. Lines 25–27 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

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



Line 30 attempts to access an element outside the array's bounds. When the member function `at` encounters an out-of-bounds index, it generates a runtime error known as an **exception**. In this program, which we compiled with GNU `g++` and ran on Linux, line 30 resulted in the following runtime error message then the program terminated:

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

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 index is out of bounds. In GNU's implementation of `array`'s `at` member function, `__n` represents the element's index, and `_Nm` represents the array's size. Section 6.15 introduces how to use exception handling to deal with runtime errors. Chapter 12 covers exception handling in depth.



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 overflow**³) can corrupt a program's data in memory and crash a program. Attackers can sometimes exploit buffer overflows by overwriting the program's executable code with malicious code.

2. C++ Standard, “17.2.4 Sizes, Alignments, and Offsets.” Accessed April 14, 2023. <https://timsong-cpp.github.io/cppwp/n4861/support.types#layout-3>.

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



Checkpoint

1 (*True/False*) C++ provides automatic array bounds checking to prevent you from referring to an array element that does not exist. Thus, an executing program cannot “walk off” either end of an array without warning.

Answer: False. Actually, 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.

2 (*Code*) Assume the array `numbers` exists and contains `int` values. What does the following code do?

```
for (size_t i{0}; i < numbers.size(); ++i) {
    numbers[i] = 1; // set element at location i to 0
}
```

Answer: The code sets each of `numbers`' elements to 1.

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 **initializer lists** to initialize an array of `ints` with five values (line 8) and an array of `doubles` with four values (line 18) and displays each array's contents:

- Line 8 explicitly tells the compiler the array's element type (`int`) and size (5).
- Line 18 lets the compiler determine the array's element type (`double`) from the initializer list's values and the array's size (4) from the number of initializers. This capability is known as **class template argument deduction (CTAD)**⁴ and simplifies your code when you know the container's initial element values in advance.

```
1 // fig06_02.cpp
2 // Initializing an array in a declaration.
3 #include <format>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8     std::array<int, 5> values{32, 27, 64, 18, 95}; // braced initializer
9
10    // output each array element
11    for (size_t i{0}; i < values.size(); ++i) {
12        std::cout << std::format("{} ", values.at(i));
13    }
14
15    std::cout << "\n\n";
```

Fig. 6.2 | Initializing an array in a declaration. (Part 1 of 2.)

4. “Class Template Argument Deduction (CTAD).” Accessed April 14, 2023. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```

16 // using class template argument deduction to determine values
17 std::array values2{1.1, 2.2, 3.3, 4.4};
18
19 // output each array element
20 for (size_t i{0}; i < values2.size(); ++i) {
21     std::cout << std::format("{} ", values2.at(i));
22 }
23
24 std::cout << "\n";
25 }
26 }
```

```

32 27 64 18 95
1.1 2.2 3.3 4.4

```

Fig. 6.2 | Initializing an array in a declaration. (Part 2 of 2.)

Fewer Initializers Than array Elements

If there are fewer initializers than array elements, the remaining array elements are **value initialized**—fundamental numeric types are set to 0, bools are set to `false`, and as we'll see in Chapter 9, objects receive the default initialization specified by their class definitions. For example, the following initializes an `int` array's elements to zero

```
std::array<int, 5> values{}; // initialize elements to 0
```

because there are fewer initializers (none in this case) than array elements. This technique also can be used to “reinitialize” an existing array's elements at execution time, as in

```
values = {};// set all elements of values to 0
```

More Initializers Than array Elements

When you explicitly state the array's size and use an initializer list, the number of initializers must be less than or equal to the size. The array declaration

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

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



Checkpoint

- 1** (*Fill-in*) An array's elements can be initialized in an array declaration by following the array name with a _____.

Answer: brace-delimited, comma-separated list of initializers.

- 2** (*Fill-in*) The compiler can use _____ to determine an array's element type and size from the values in an initializer list.

Answer: class template argument deduction (CTAD).

- 3** (*Code*) Initialize the `int` array named `grades` with the values 80, 97 and 75. Let the compiler use CTAD to infer the array's element type.

Answer: `std::array grades{80, 97, 75};`

6.6 Range-Based for Statement

It's common to process all the elements of an array. The **range-based for statement** allows you to do this without using a counter. When processing all elements of an **array**, use the **range-based for statement** to ensure that your code does not “step outside” the **array’s bounds**. At the end of this section, we’ll compare the counter-controlled **for** and range-based **for** statements. Figure 6.3 uses the range-based **for** to display an array’s contents (lines 12–14 and 23–25) and multiply each element’s value by 2 (lines 17–19).



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

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10

calculating a running total of items
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 `item` in `items`,” perform some work. In each iteration, the loop assigns the next `items` element to the `const int` reference `item`, then executes the loop’s body—`item` refers to one element’s value 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.⁵ Here we declare the range variable `item` as a `const` reference. Recall that a reference is an alias for another variable in memory—in this case, one of the array’s elements.



Declaring a `range` variable as a `const` reference can improve performance. A reference prevents the loop from copying each value into the range variable. This is particularly important when manipulating large objects.

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 `item` is an `int&`—that is, a reference to an `int`. This is not a `const` reference, so any change you make to `item` changes the corresponding array element’s value.

C++20 Range-Based for with Initializer

C++20 adds the `range-based for with initializer` statement. The initializer in line 29 defines `runningTotal` and sets it to 0, then lines 29–33 calculate the running total of `items`’ elements. The variable `runningTotal` exists only until the loop terminates.

Avoiding Indices

In most cases, a range-based for statement can be used instead of the counter-controlled for statement. For example, totaling the integers in an array requires access only to the element values—their index positions in the array are irrelevant.

External vs. Internal Iteration

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

```
for (size_t counter{0}; counter < items.size(); ++counter) {
    std::cout << std::format("{} ", 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 the control variable `counter` incorrectly.

Each of these could result in accessing elements outside the array `items`’ bounds.

5. You can use the range-based for statement with most of the C++ standard library’s containers, which we discuss in Chapter 13.

The range-based `for` statement uses **internal iteration**, hiding the iteration details from you. You specify *what* array the range-based `for` should process, and it knows *how* to get each value from the array and stop iterating when there are no more values.



Checkpoint

- 1 (*True/False*) When processing all elements of an array, use the range-based `for` statement because it ensures that your code does not “step outside” the array’s bounds.

Answer: True.

- 2 (*True/False*) Variables defined in a range-based `for`’s initializer exist after the loop terminates.

Answer: False. Actually, variables defined in a range-based `for`’s initializer exist only until the loop terminates.

- 3 (*Code*) The following counter-controlled `for` statement outputs the `ints` in the array `grades`.

```
for (int counter{0}; counter < grades.size(); ++counter) {  
    std::cout << std::format("{} ", grades[counter]);  
}
```

Convert this counter-controlled `for` statement to a range-based `for` statement.

Answer:

```
for (const int& grade : grades) {  
    std::cout << std::format("{} ", grade);  
}
```

6.7 Calculating array Element Values and an Intro to `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). Line 14 generates values by multiplying each successive value of the loop counter, `i`, by 2 and adding 2.

```
1 // fig06_04.cpp  
2 // Set array values to the even integers from 2 to 10.  
3 #include <iostream>  
4 #include <array>  
5 #include <array>  
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     std::array<int, arraySize> values{}; // array values has 5 elements  
12  
13     for (int i{0}; i < values.size(); ++i) { // set the values  
14         values.at(i) = 2 + 2 * i;  
15     }
```

Fig. 6.4 | Set array `values` to the even integers from 2 to 10. (Part I of 2.)

```

16 // output contents of array values in tabular format
17 for (const int& value : values) {
18     std::cout << std::format("{} ", value);
19 }
20
21     std::cout << '\n';
22 }
23 }
```

```
2 4 6 8 10
```

Fig. 6.4 | Set array values to the even integers from 2 to 10. (Part 2 of 2.)

Constants

In Chapter 5, we used the `const` qualifier to indicate that a variable's value does not change after it's initialized. Sometimes, compilers will optimize your code for better performance by initializing `const` variables at compile time; however, that's not guaranteed. The `constexpr` qualifier also can be used to declare a constant variable. Such a variable is implicitly `const` and is explicitly **initialized at compile-time**, eliminating the possibility of runtime initialization.

Defining an array's size as a constant instead of a literal value makes programs clearer and easier to update. This technique eliminates **magic numbers**—literal numeric values with no context to help 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.

Cannot Modify a `constexpr` Variable After It's Initialized

Line 9 uses `constexpr` to declare the constant `arraySize` with the value 5. Attempting to modify `arraySize` after it's initialized, as in

```
arraySize = 7;
```



results in a compilation error:⁶

- Visual C++:


```
error C3892:
        that is const
```
- g++:


```
error: assignment of read-only variable
```
- clang++:


```
error: cannot assign to variable
        const-qualified type
```

6. 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.”



Modern C++: Doing More at Compile-Time

A modern C++ theme is to do more at compile-time for better runtime performance. `constexpr` is one of several features we'll present that eliminate runtime overhead and improve performance. Each C++ standard since C++11 has expanded `constexpr`'s use.

In Section 15.13, you'll see that `constexpr` can be applied to functions. When you call such a function with `constexpr` arguments, the compiler can determine the call's final result and replace the call with a constant, eliminating the runtime function-call overhead and thus improving performance. Many functions throughout the C++ standard library are now declared `constexpr` to take advantage of this optimization.

Going forward, we'll define each variable whose value can be determined at compile-time as `constexpr`. When we introduce `constexpr` functions in Section 15.13, we'll demonstrate compile-time evaluation of `constexpr` function calls.



Checkpoint

- 1 *(Fill-in)* The _____ qualifier enables you to declare variables that can be evaluated at compile-time and result in a constant.

Answer: `constexpr`.

- 2 *(True/False)* A `const` variable must be initialized at compile time, but a `constexpr` variable can be initialized at execution time.

Answer: False. Actually, a `constexpr` variable must be initialized at compile time, but a `const` variable can be initialized at execution time.

6.8 Totaling array Elements

Often, array elements represent a series of values for use in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the array elements and then calculate the class average for the exam. Processing a collection of values into a single value is known as **reduction**—a common functional-style programming operation we discuss in Section 6.14. Figure 6.5 uses a range-based `for` statement (lines 12–14) to total the values in a four-element array of `ints`. Section 6.14 shows how to perform this calculation using the C++ standard library's `accumulate` function.

```
1 // fig06_05.cpp
2 // Compute the sum of an array
3 #include <iostream>
4 #include <array>
5 #include <format>
6
7 int main() {
8     std::array items{10, 20, 30, 40}; // type inferred as array<int, 4>
9     int total{0};
10 }
```

Fig. 6.5 | Compute the sum of an array (Part 1 of 2.)

```

11 // sum the contents of items
12 for (const int& item : items) {
13     total += item;
14 }
15
16 std::cout << std::format("Total of array elements: {}\n", total);
17 }
```

Total of array elements: 100

Fig. 6.5 | Compute the sum of an array (Part 2 of 2.)



Checkpoint

1 (*Fill-in*) Processing a collection of values into a single value is known as _____.

Answer: reduction.

2 (*Code*) Given the int array named `grades` with the values 80, 97 and 75, what does this code do?

```

double total{0.0};

for (const int& grade : grades) {
    total += grade;
}

std::cout << total / grades.size() << '\n';
```

Answer: This code calculates and displays the average of the three grades.

6.9 Using a Primitive Bar Chart to Display array Data Graphically

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

A professor might graph the number of exam grades in 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 values. In this example, we initialize the array `frequencies` with frequency values.

Figure 6.6 reads the numbers from `frequencies` and graphs them as a bar chart, displaying each grade range followed by a bar of asterisks indicating the number of grades in that range. The program operates as follows:

- Line 8 declares the array with `constexpr` because the program never modifies `frequencies`, and its values are known at compile-time.
- To label each bar, lines 15–21 output a grade range (e.g., "70-79: ") based on the current value of `i`. In lines 16–17, the format specifier `:02d` indicates that an integer (represented by `d`) should be formatted using a field width of 2 and a leading 0 if the integer is fewer than two digits. In line 20, the format specifier `:>5d` indicates that an integer should be right-aligned (`>`) in a field width of 5.
- The nested `for` statement (lines 26–28) outputs the current bar's asterisks. The loop-continuation condition in line 26 (`stars < frequency`) enables the inner `for` loop to count from 0 up to `frequency`. Thus, a value from `frequencies` determines the number of asterisks to display. In this example, `frequencies` 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.

```
1 // fig06_06.cpp
2 // Printing a student grade distribution as a primitive bar chart.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6
7 int main() {
8     constexpr std::array frequencies{0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
9
10    std::cout << "Grade distribution:\n";
11
12    // for each element of frequencies, output a bar of the chart
13    for (int i{0}; const int& frequency : frequencies) {
14        // output bar labels ("00-09:", ..., "90-99:", "100:")
15        if (i < 10)
16            std::cout << std::format("{:02d}-{:02d}: ", 
17            i * 10, (i * 10) + 9);
18        else {
19            std::cout << std::format("{:>5d}: ", 100);
20        }
21
22        ++i;
23
24        // print bar of asterisks
25        for (int stars{0}; stars < frequency; ++stars) {
26            std::cout << '*';
27        }
28
29        std::cout << '\n'; // start a new line of output
30    }
31 }
32 }
```

Fig. 6.6 | Printing a student grade distribution as a primitive bar chart. (Part 1 of 2.)

```

Grade distribution:
00-09:
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. (Part 2 of 2.)

6.10 Using array Elements as Counters

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 frequencies of each die face as the program rolled the die 60,000,000 times. Figure 6.7 reimplements that simulation using an array of frequency counters.

```

1 // fig06_07.cpp
2 // Die-rolling program using an array instead of switch.
3 #include <format>
4 #include <iostream>
5 #include <array>
6 #include <random>
7
8 int main() {
9     // set up random-number generation
10    std::random_device rd; // used to seed the default_random_engine
11    std::default_random_engine engine{rd()}; // rd() produces a seed
12    std::uniform_int_distribution randomDie{1, 6};
13
14    constexpr size_t arraySize{7}; // ignore element zero
15    std::array<int, arraySize> frequency{}; // initialize to 0s
16
17    // roll die 60,000,000 times; use die value as frequency index
18    for (int roll{1}; roll <= 60'000'000; ++roll) {
19        ++frequency.at(randomDie(engine));
20    }
21
22    std::cout << std::format("{}{:>13}\n", "Face", "Frequency");
23
24    // output each array element
25    for (size_t face{1}; face < frequency.size(); ++face) {
26        std::cout << std::format("{:>4}{:>13}\n", face, frequency.at(face));
27    }
28}

```

Fig. 6.7 | Die-rolling program using an array instead of switch. (Part 1 of 2.)

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. (Part 2 of 2.)

Figure 6.7 uses the array `frequency` (line 15) to count the occurrences of die values. Line 19 replaces the `switch` statement in lines 23–45 of Fig. 5.3. It uses a random die value as an index to determine which `frequency` element to increment for each die roll. The dot (.) operator in line 19 has higher precedence than the `++` operator, so the statement selects a `frequency` element, then increments its value.

The call `randomInt(engine)` produces a random index 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 49–54 of Fig. 5.3 by looping through array `frequency` to output the results (Fig. 6.7, lines 25–27). We used a counter-controlled `for` statement here rather than a range-based `for` statement, so we could skip printing element 0 of `frequency`.



Checkpoint

- I (Code) Rewrite lines 22–27 of Fig. 6.7 to display the columns left-aligned, as in:

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

Answer:

```
std::cout << std::format("{:<8}{}\n", "Face", "Frequency");

// output each array element
for (size_t face{1}; face < frequency.size(); ++face) {
    std::cout << std::format("{:<8}{}\n", face, frequency.at(face));
}
```

6.11 Using arrays to Summarize Survey Results

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

Twenty students were asked to rate the food quality in the student cafeteria on a scale of 1 to 5, with 1 being “awful” and 5 being “excellent.” Place the 20 responses in an array of int values 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 9–10) is a 20-element array of `ints` initialized with the students' survey responses. It's declared `constexpr` because its values do not (and should not) change and are known at compile-time. We use a six-element array `frequency` (line 18) to count each response's number of occurrences. Each `frequency` element 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.

```

1 // fig06_08.cpp
2 // Poll analysis program.
3 #include <format>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8     // place survey responses in array responses
9     constexpr std::array responses{
10         1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
11
12     // initialize frequency counters to 0
13     constexpr size_t frequencySize{6}; // size of array frequency
14     std::array<int, frequencySize> frequency{};
15
16     // for each response in responses, use that value
17     // as frequency index to determine element to increment
18     for (const int& response : responses) {
19         ++frequency.at(response);
20     }
21
22     std::cout << std::format("{}{:>12}\n", "Rating", "Frequency");
23
24     // output each array element
25     for (size_t rating{1}; rating < frequency.size(); ++rating) {
26         std::cout << std::format("{:>6}{:>12}\n",
27             rating, frequency.at(rating));
28     }
29 }
```

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

Fig. 6.8 | Poll analysis program.

The first `for` statement (lines 18–20) takes one response at a time from `responses` (each is a value in the range 1–5) and increments one of the counters `frequency.at(1)` to `frequency.at(5)`. The key statement is line 19, which increments the appropriate counter depending on the value of `responses.at(response)`. Regardless of the number of responses processed in the survey, the program requires only a six-element array (ignor-

ing 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

Sorting data 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 computer science. Often it may be necessary to determine whether an array contains a value that matches a particular key value. The process of finding a key value is called searching. This section uses the built-in C++ standard library `sort` function to arrange the elements in an array into ascending order and the built-in `binary_search` function to determine whether a value is in a sorted array.

Demonstrating Functions `sort` and `binary_search`

Figure 6.9 creates an unsorted array of seven `string` objects named `colors` (lines 13–14). The `using` declaration in line 10 enables us to initialize this array using `string` object literals. You form a `string` object literal by placing an `s` after the string's closing quote. For example, in "red"s, the `s` following the literal tells the compiler that this literal is a `std::string` object. Using `string` object literals here enables the compiler to infer from the initializers that the array's element type is `std::string`. Lines 18–20 display `color`'s contents. Note that the loop's control variable is declared as a

```
const std::string&
```

Using a reference for the range variable `color` prevents the loop from copying the current `string` object into `color` during each iteration. In programs processing large numbers of strings, that copy operation would degrade performance.

```

1 // fig06_09.cpp
2 // Sorting and searching arrays.
3 #include <array>
4 #include <algorithm> // contains sort and binary_search
5 #include <format>
6 #include <iostream>
7 #include <string>
8
9 int main() {
10     using namespace std::string_literals; // enables string object literals
11
12     // colors is inferred to be an array<string, 7>
13     std::array colors{"red"s, "orange"s, "yellow"s,
14                      "green"s, "blue"s, "indigo"s, "violet"s};
15
16     // output original array
17     std::cout << "Unsorted colors array:\n    ";
18     for (const std::string& color : colors) {
19         std::cout << std::format("{} ", color);
20     }

```

Fig. 6.9 | Sorting and searching arrays. (Part I of 2.)

```

21
22    // sort contents of colors
23    std::sort(std::begin(colors), std::end(colors));
24
25    // output sorted array
26    std::cout << "\nSorted colors array:\n  ";
27    for (const std::string& color : colors) {
28        std::cout << std::format("{} ", color);
29    }
30
31    // search for "indigo" in colors
32    bool found{std::binary_search(
33        std::begin(colors), std::end(colors), "indigo")};
34    std::cout << std::format("\n\n\"indigo\" {} found in colors array\n",
35    found ? "was" : "was not");
36
37    // search for "cyan" in colors
38    found = std::binary_search(
39        std::begin(colors), std::end(colors), "cyan");
40    std::cout << std::format("\n\"cyan\" {} found in colors array\n",
41    found ? "was" : "was not");
42 }

```

```

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. (Part 2 of 2.)

Next, line 23 uses the C++ standard library function `sort` (from header `<algorithm>`) to place the elements of the array `colors` into ascending order. For strings, this is a **lexicographical sort**—that is, the strings are ordered by their characters’ numerical values in the underlying character set. The `sort` function’s arguments specify the range of elements to sort—in this case, the entire array. The arguments `std::begin(colors)` and `std::end(colors)` return “iterators” that represent the array’s beginning and end, respectively. Chapter 13 discusses iterators in depth. Functions `begin` and `end` are defined in the `<array>` header. As you’ll see in later chapters, `sort` can sort the elements of several kinds of data structures. Lines 27–29 display the sorted array’s contents.

Lines 32–33 and 38–39 use C++ standard library function `binary_search` (from header `<algorithm>`) to determine whether a value is in the array. **The sequence of values must first 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 is the search key—the value to find in the array. The function returns a `bool` indicating whether the value was found. In Chapter 14, we’ll use the C++ Standard function `find` to obtain a search key’s index in an array.



Checkpoint

- 1** (*Fill-in*) The arguments `std::begin(colors)` and `std::end(colors)` return _____ that represent the `colors` array's beginning and end, respectively.

Answer: iterators.

- 2** (*True/False*) Standard library function `binary_search` (from header `<algorithm>`) can determine whether a value is in a sorted array.

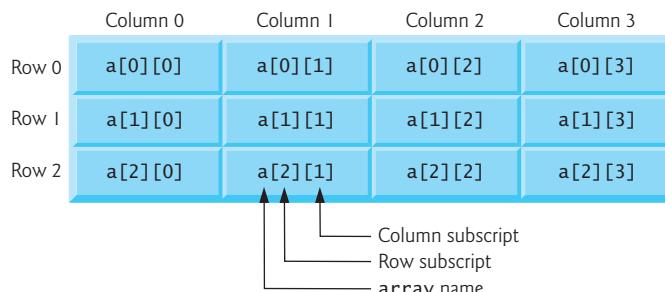
Answer: True.

- 3** (*True/False*) Performing a `binary_search` on an unsorted array is a syntax error that could lead to incorrect results.

Answer: False. Actually, performing a `binary_search` on an unsorted array is a logic error that could lead to incorrect results.

6.13 Multidimensional arrays

You can use arrays with two dimensions (i.e., indices) to represent **tables of values** with data 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 named `a`:



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[row][column]`. Similarly, you can access each element with `at`, as in

```
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.

Figure 6.10 demonstrates initializing two-dimensional arrays in declarations. Lines 11–12 each create an array of arrays with two rows and three columns.

```

1 // fig06_10.cpp
2 // Initializing multidimensional arrays.
3 #include <iostream>
4 #include <array>
5
6 constexpr size_t rows{2};
7 constexpr size_t columns{3};
8 void printArray(const std::array<std::array<int, columns>, rows>& a);
9
10 int main() {
11     constexpr std::array values1{std::array{1, 2, 3}, std::array{4, 5, 6}};
12     constexpr std::array values2{std::array{1, 2, 3}, std::array{4, 5, 0}};
13
14     std::cout << "values1 by row:\n";
15     printArray(values1);
16
17     std::cout << "\nvalues2 by row:\n";
18     printArray(values2);
19 }
20
21 // output array with two rows and three columns
22 void printArray(const std::array<std::array<int, columns>, rows>& a) {
23     // loop through array
24     for (const auto& row : a) {
25         // loop through columns of current row
26         for (const auto& element : row) {
27             std::cout << element << ' ';
28         }
29
30         std::cout << '\n'; // start new line of output
31     }
32 }
```

```
values1 by row:
1 2 3
4 5 6
```

```
values2 by row:
1 2 3
4 5 0
```

Fig. 6.10 | Initializing multidimensional arrays.

Declaring an array of arrays

In lines 11–12, the compiler infers that `values1` and `values2` are arrays of arrays with two rows of three columns each and in which the elements are type `int`. Consider the two initializers for `values1`:

```
std::array{1, 2, 3}
std::array{4, 5, 6}
```

From these, the compiler infers that `values1` has two rows. Each of these initializers creates an array of three elements, so the compiler infers each row of `values1` has three col-

umns. Finally, the values in these row initializers are `ints`, so the compiler infers that `values1`'s element type is `int`.

Displaying an array of arrays

The program calls the function `printArray` to output each array's elements. The function prototype (line 8) and definition (lines 22–32) specify that `printArray` receives a two-row and three-column array of `ints` via a `const` reference. In the type

```
const std::array<std::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 elements is an array of `ints` containing `columns` (3) elements. The parameter receives the array as a `const` reference because `printArray` does not modify the elements.

Nested Range-Based for Statements

To process the elements of a two-dimensional array, we use a nested loop:

- 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 24 and 26 introduce the `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` is a reference to a `const 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 27 displays the `element` 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:

```
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) << ' '; // or a[row][column]
    }
    cout << '\n';
}
```

Initializing an array of arrays with a Fully Braced Initializer List

If you specify the array's dimensions explicitly when you declare it, you can simplify its initializer list. For example, line 11 could be written as

```
constexpr std::array<std::array<int, columns>, rows> values1{
    {{1, 2, 3}, // row 0
     {4, 5, 6}} // row 1
};
```

In this case, if an initializer sublist has fewer elements than the number of columns, the row's remaining elements would be value initialized.

Common Two-Dimensional array Manipulations: Setting a Row's Values

Let's consider several other common manipulations using the three-row and four-column array `a` from the diagram at the beginning of Section 6.13. The following `for` statement sets all the elements in row 2 to zero:

```
for (size_t column{0}; column < a.at(2).size(); ++column) {
    a.at(2).at(column) = 0; // or 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:

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

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`:

```
int 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]
    }
}
```

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:

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

mdarray

The C++ Standard Committee is working on a true multidimensional array container called `mdarray` for future C++ versions. You can follow the progress on `mdarray` and the related type `mdspan` at

- `mdarray`—<https://isocpp.org/files/papers/D1684R0.html>
- `mdspan`—<https://wg21.link/P0009>

**Checkpoint**

1 (*Fill-in*) An array with m rows and n columns is called a(n) _____.

Answer: m -by- n array.

2 (*Fill-in*) The _____ keyword tells the compiler to infer (determine) a variable's data type based on the variable's initializer value.

Answer: `auto`.

3 (*Code*) Given a 3-by-3 array of `ints` named `table`, use a counter-controlled `for` statement to set each `table` element to the sum of its subscripts.

Answer:

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

6.14 Intro to Functional-Style Programming

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 that’s easier to read, debug and modify, and has fewer errors. 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`:

```
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` to 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 compute the sum of a range’s values, reducing them to a single value. This is known as a **reduction** and is a common functional-style programming operation.

```

1 // fig06_11.cpp
2 // Compute the sum of the elements of an array using accumulate.
3 #include <array>
4 #include <format>
5 #include <iostream>
6 #include <numeric>
7
8 int main() {
9     constexpr std::array integers{10, 20, 30, 40};
10    std::cout << std::format("Total of array elements: {}\n",
11        std::accumulate(std::begin(integers), std::end(integers), 0));
12 }
```

Total of array elements: 100

Fig. 6.11 | Compute the sum of the elements of an array using `accumulate`.

Like function `sort` in Section 6.12, function `accumulate`’s first two arguments (line 11) specify the range of elements to sum—in this case, the elements from the beginning to the end of `integers`. The function *internally* adds to the running total of the elements it processes, hiding those calculations. The third argument is the running total’s initial value (0), and this argument’s type is `accumulate`’s return type. You’ll see momentarily how to customize `accumulate`’s reduction.

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 and letting the library determine *how* to do it is known as **declarative programming**—another hallmark 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** and are commonly used in functional programming. Consider function `accumulate`, which totals elements by default. It also provides an overload, which receives as its fourth argument a function that defines how to perform the reduction. Rather than simply totaling the values, Fig. 6.12 calculates the product of the values.

```

1 // fig06_12.cpp
2 // Compute the product of an array
3 #include <array>
```

Fig. 6.12 | Compute the product of an array’s elements using `accumulate`. (Part 1 of 2.)

```

4 #include <format>
5 #include <iostream>
6 #include <numeric>
7
8 int multiply(int x, int y) {
9     return x * y;
10 }
11
12 int main() {
13     constexpr std::array integers{1, 2, 3, 4, 5};
14
15     std::cout << std::format("Product of integers: {}\n", std::accumulate(
16         std::begin(integers), std::end(integers), 1, multiply));
17
18     std::cout << std::format("Product of integers with a lambda: {}\n",
19         std::accumulate(std::begin(integers), std::end(integers), 1,
20             [] (const auto& x, const auto& y){return x * y;}));
21 }
```

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

Fig. 6.12 | Compute the product of an array's elements using `accumulate`. (Part 2 of 2.)

Calling `accumulate` with a Named Function

Lines 15–16 call `accumulate` for the array `integers` (defined at line 13). We're calculating the product, 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 `integers`, 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, specified as `accumulate`'s third argument) 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.
- Finally, the last call passes 24 and the array's fifth element (5), producing the overall result 120, which `accumulate` returns to its caller.

Calling accumulate with a Lambda Expression

Sometimes, you do not need to reuse a function, in which case, you can define a function where it's needed by using a **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 19–20 uses the following lambda expression to perform the same task as `multiply`:

```
[](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.

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 type based on the context in which the lambda is called. In this example, `accumulate` calls the lambda once for each array element, 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`. The preceding lambda is a **generic lambda**—it uses `auto` to infer each parameter's type. 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 cannot modify the caller's variables.
- They are **references for performance to ensure that if the lambda is used with large objects, it does not copy them.**

This lambda could be used with any type that supports the `*` operator. Chapter 14 discusses lambda expressions in detail.

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** (header `<ranges>`) makes functional-style programming more convenient. Here, we introduce two key aspects of this library—ranges and views:

- 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.

Figure 6.13, which we've broken into pieces for discussion purposes, demonstrates several functional-style operations using C++20 ranges. We'll cover more in Chapter 14. At the time of this writing, this example does not compile in Xcode.

showValues Lambda for Displaying This Application's Results

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. Instead, we defined a generic lambda (lines 12–20) to show that you can store a lambda in a local variable (`showValues`; lines 11–21). Then, you can use the variable's name to call the lambda, as we do in lines 24, 29, 34, 40 and 51.

```

1 // fig06_13.cpp
2 // Functional-style programming with C++20 ranges and views.
3 #include <array>
4 #include <format>
5 #include <iostream>
6 #include <numeric>
7 #include <ranges>
8
9 int main() {
10     // Lambda to display results of range operations
11     auto showValues{
12         [](auto& values, const std::string& message) {
13             std::cout << std::format("{}: ", message);
14
15             for (const auto& value : values) {
16                 std::cout << std::format("{} ", value);
17             }
18
19             std::cout << '\n';
20         }
21     };
22 }
```

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

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

In many of this chapter's examples, we created an array, then processed its values. This required preallocating the array with the appropriate number of elements. Sometimes, 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 23 uses the `<ranges>` library's `views::iota` to generate a range of integers from its first argument (1) up to, but not including, its second argument (11). This is known as a **half-open range**. The values are not generated until the program iterates over the results, such as when we call `showValues` (line 24) to display the values.



```

23     auto values1{std::views::iota(1, 11)}; // generate integers 1-10
24     showValues(values1, "Generate integers 1-10");
25 }
```

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, using an `if` statement to check whether each element matches a condition. You could then do something with that element, such as add it to a container. That requires explicitly defining an iteration control statement and mutable variables, which can be error-prone, as discussed in Section 6.6.

With ranges and views, we can use `views::filter` to focus on *what* we want to accomplish—in this case, getting the even integers in the range 1–10. The `values2` initializer in lines 27–28 uses the `| operator` to connect multiple operations. The first operation (`values1` from line 23) 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 `|`.

```

26 // filter each value in values1, keeping only the even integers
27 auto values2{values1 |
28     std::views::filter([](const auto& x) {return x % 2 == 0;})};
29 showValues(values2, "Filtering even integers");
30

```

```
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. We passed a lambda that returns `true` if its argument is divisible by 2.

After lines 27–28 execute, `values2` represents a **lazy 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 do 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.

However, the pipeline is lazy—no results are produced until you iterate over `values2`.

When `showValues` iterates over `values2`, `views::iota` produces a value, then `views::filter` calls its function argument to determine whether to keep the value. If that function returns `true`, 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 range being mapped. With C++20 ranges, `views::transform` performs mapping operations. The pipeline in lines 32–33 adds another operation to the pipeline from lines 27–28, mapping the filtered results from `values2` to their squares with the lambda expression passed to `views::transform` in line 33.

```

31 // map each value in values2 to its square
32 auto values3{
33     values2 | std::views::transform([](const auto& x) {return x * x;});
34 showValues(values3, "Mapping even integers to squares");
35

```

```
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:

1. `views::iota` produces a value.
2. `views::filter` determines whether the value is even. If so, the value is passed to *Step 3*. Otherwise, processing proceeds with the next value generated by `views::iota` in *Step 1*.
3. `views::transform` calculates the square of the even integer (as specified by line 33's lambda), then the range-based `for` loop in `showValues` displays the value and processing proceeds with the next value generated by `views::iota` in *Step 1*.

Combining Filtering and Mapping Operations into a Pipeline

A pipeline may contain an arbitrary number of operations separated by `|` operators. The pipeline in lines 37–39 combines the preceding operations into a single pipeline, and line 40 displays the results.

```
36     // combine filter and transform to get squares of the even integers
37     auto values4{
38         values1 | std::views::filter([](const auto& x) {return x % 2 == 0;})
39             | std::views::transform([](const auto& x) {return x * x;});
40     showValues(values4, "Squares of even integers");
41 }
```

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

Reducing a Range Pipeline with `accumulate`

C++ standard library functions like `accumulate` also work with lazy range pipelines. Line 44 performs a reduction that sums the squares of the even integers produced by the pipeline in lines 37–39.

```
42     // total the squares of the even integers
43     std::cout << std::format("Sum squares of even integers 2-10: {}\n",
44         std::accumulate(std::begin(values4), std::end(values4), 0));
```

```
Sum squares of even integers 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 47 creates an array containing 1–10, then uses it in a pipeline that calculates the squares of the even integers in the array.

```

46 // process a container
47 constexpr std::array numbers{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
48 auto values5{
49     numbers | std::views::filter([](const auto& x) {return x % 2 == 0;})
50     | std::views::transform([](const auto& x) {return x * x;});
51     showValues(values5, "Squares of even integers in array numbers");
52 }

```

Squares of even integers in array numbers: 4 16 36 64 100



Checkpoint

1 (*Fill-in*) Functions that receive other functions as arguments are called _____ and are commonly used in functional programming.

Answer: higher-order functions.

2 (*Fill-in*) A view enables you to specify an operation that manipulates a range. Views are _____—you can chain them together to process a range’s elements through multiple operations.

Answer: composable.

3 (*Fill-in*) The `<ranges>` library’s `views::iota` generates a range of integers from its first argument up to, but not including, its second argument—known as a _____ range.

Answer: half-open.

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`⁷ (from header `<vector>`) “out for a spin” (Fig. 6.14). A `vector` is similar to an `array` but supports dynamic resizing. 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 by detecting and handling an out-of-bounds `vector` index. At that point, we’ll discuss the `<stdexcept>` header (line 5). Lines 7–8 are the function prototypes for functions `outputVector` (lines 93–99) and `inputVector` (lines 102–106), which display a `vector`’s contents and input values into a `vector`, respectively.

```

1 // fig06_14.cpp
2 // Demonstrating C++ standard library class template vector.
3 #include <iostream>
4 #include <vector>
5 #include <stdexcept>
6
7 void outputVector(const std::vector<int>& items); // display the vector
8 void inputVector(std::vector<int>& items); // input values into the vector
9

```

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

7. Chapter 13 discusses more `vector` capabilities.

Creating vector Objects

Lines 11–12 create two `vector` objects that store values of type `int`—`integers1` contains seven elements, and `integers2` contains 10 elements. By default, each `vector` object's elements are set to 0. Like arrays, `vectors` can store most data types—simply replace `int` in `std::vector<int>` with the appropriate type.

```
10 int main() {  
11     std::vector<int> integers1(7); // 7-element vector<int>  
12     std::vector<int> integers2(10); // 10-element vector<int>  
13 }
```

Note that we used parentheses rather than a braced initializer 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 value 7, not a seven-element `vector`:

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

If you know the `vector`'s contents at compile-time, you can use an initializer list and class template argument deduction (CTAD) to define the `vector`, as we've done with `arrays`. For example, the following statement defines a four-element `vector` of `doubles`:

```
std::vector integers1{1.1, 2.2, 3.3, 4.4};
```

vector Member Function `size`; Function `outputVector`

Line 15 uses `vector` member function `size` to obtain `integers1`'s number of elements.⁸ Line 17 passes `integers1` to function `outputVector` (lines 93–99), which displays the `integers2`.

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

```
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
```

8. You also can use the global function `std::size`, as in `std::size(integers1)`.

Function `inputVector`

Lines 26–27 pass `integers1` and `integers2` to function `inputVector` (lines 102–106) to read values for each vector's elements from the user.

```

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

```

```

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 defines overloaded operators `==` and `!=` that compare two arrays or two vectors for equality or inequality, respectively. Line 38 compares two vector objects using the `!=` operator, returning `true` if the contents of two vectors are not equal and `false` otherwise. Two vectors are not equal if they have different lengths or the elements at the same indexes are not equal

```

35 // use inequality (!=) operator with vector objects
36 std::cout << "\nEvaluating: integers1 != integers2\n";
37
38 if (integers1 != integers2) {
39     std::cout << "integers1 and integers2 are not equal\n";
40 }
41

```

```

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

```

Initializing One vector with the Contents of Another

You can initialize a new vector by copying the contents of an existing one. Line 44 creates a vector object `integers3` and initializes it with a copy of `integers1`. Here, we used CTAD to infer `integers3`'s element type from that of `integers1`. Line 44 invokes the class template vector's copy constructor to perform the copy operation. You'll learn about copy constructors in detail in Chapter 11. Lines 46–48 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

```
42 // create vector integers3 using integers1 as an
43 // initializer; print size and contents
44 std::vector integers3{integers1}; // copy constructor
45
46 std::cout << "\nSize of vector integers3 is " << integers3.size()
47     << "\nvector after initialization: ";
48 outputVector(integers3);
49
```

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

Assigning vectors and Comparing vectors for Equality

Line 52 assigns `integers2` to `integers1`, demonstrating that the assignment (`=`) operator is overloaded to work with `vector` objects. Lines 54–57 output the contents of both objects to show that they now contain identical values. Line 62 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.

```
50 // use overloaded assignment (=) operator
51 std::cout << "\nAssigning integers2 to integers1:\n";
52 integers1 = integers2; // assign integers2 to integers1
53
54 std::cout << "integers1: ";
55 outputVector(integers1);
56 std::cout << "integers2: ";
57 outputVector(integers2);
58
59 // use equality (==) operator with vector objects
60 std::cout << "\nEvaluating: integers1 == integers2\n";
61
62 if (integers1 == integers2) {
63     std::cout << "integers1 and integers2 are equal\n";
64 }
65
```

```
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 67 and 71 use the `vector` member function `at` to obtain an element and use it to get a value from the `vector` and replace a value in the `vector`, respectively. If the index is valid, `at` returns either

- a reference to that element, which can be used to change the `vector` element's value, or

- a `const` reference to that element, which can be used to read but not change the element’s value.

The `at` function returns a `const` reference if it’s called on a `const` `vector` or via a `const` reference.

```

66 // use the value at location 5 as an rvalue
67 std::cout << "\nintegers1.at(5) is " << integers1.at(5);
68
69 // use integers1.at(5) as an lvalue
70 std::cout << "\n\nAssigning 1000 to integers1.at(5)\n";
71 integers1.at(5) = 1000;
72 std::cout << "integers1: ";
73 outputVector(integers1);
74

```

```

integers1.at(5) is 13
Assigning 1000 to integers1.at(5)
integers1: 8 9 10 11 12 1000 14 15 16 17

```

Like arrays, `vectors` have a `[]` operator. C++ does not require bounds checking when accessing elements with `[]`. Therefore, you must ensure that operations using `[]` do not accidentally manipulate elements outside the `vector`’s bounds.

Exception Handling: Processing an Out-of-Range Index

Line 78 attempts to output the value in `integers1.at(15)`, which is outside the `vector`’s bounds. Member function `at`’s bounds checking recognizes the invalid index and **throws an exception**, which indicates a problem at execution time. The name “exception” suggests that the problem occurs infrequently. **Exception handling** enables you to create **fault-tolerant programs** that can handle (or catch) exceptions. Sometimes, this allows a program to continue executing as if no problems were encountered. For example, this program 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, requiring it to terminate after cleaning up any resources it uses (such as closing files or database connections). Here, we introduce exception handling briefly. You’ll throw exceptions from custom functions in Chapter 9, and we’ll discuss exception handling in detail in Chapter 12.

```

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

```

```

Attempt to display integers1.at(15)
An exception occurred: invalid vector 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 76–82). The **try block** (lines 76–79) contains the code that might throw an exception, and the **catch block** (lines 80–82) contains the code that handles the exception if one occurs. The braces that delimit `try` and `catch` blocks' bodies are required. As you'll see in Chapter 12, a single `try` block can have many `catch` blocks to handle different types of exceptions that might be thrown. If the code in the `try` block executes successfully, lines 80–82 are ignored.

Executing the `catch` Block

When the program calls the `vector` member function `at` with the argument 15 (line 78), 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. Bounds checking is performed at execution time, so `at` generates an exception. Specifically, line 78 throws an `out_of_range` exception (from header `<stdexcept>`) to notify the program of this problem. This immediately terminates the `try` block, skipping any remaining statements in that block. Then, the `catch` block begins executing. If you declared any variables in the `try` block, they're now out of scope and not accessible to the `catch` block.

A `catch` block should declare its exception parameter (`ex`) as a `const` reference—we'll say more about this in Chapter 12. 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.

what Member Function of the Exception Object

When lines 80–82 catch the exception, the program displays a message (which varies by compiler) indicating the problem that occurred. Line 81 calls the exception object's `what` member function to get the error message. 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 85–89 execute next.

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 its number of elements varies. To demonstrate this, line 85 shows the current size of `integers3`, line 86 calls the `vector`'s `push_back` member function to add a new element containing 1000 to the end of the `vector`, and line 87 shows the new size of `integers3`. Line 89 then displays `integers3`'s new contents.

```
84 // changing the size of a vector
85 std::cout << "\nCurrent integers3 size is: " << integers3.size();
86 integers3.push_back(1000); // add 1000 to the end of the vector
87 std::cout << "\nNew integers3 size is: " << integers3.size()
88     << "\nintegers3 now contains: ";
89 outputVector(integers3);
90 }
91
```

```
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. You could use a counter-controlled loop, but the range-based `for` is recommended. Similarly, function `inputVector` uses a range-based `for` statement with an `int&` range variable that can be used to store an input value in the corresponding vector element.

```
92 // output vector contents
93 void outputVector(const std::vector<int>& items) {
94     for (const int& item : items) {
95         std::cout << item << ' ';
96     }
97
98     std::cout << '\n';
99 }
100
101 // input vector contents
102 void inputVector(std::vector<int>& items) {
103     for (int& item : items) {
104         std::cin >> item;
105     }
106 }
```

Straight-Line Code

As you worked through this chapter’s examples, you saw lots of `for` loops. You may have noticed in our Objects-Natural sections that much of the code that creates and uses objects is straight-line sequential code with few control statements. Working with objects often “flattens” code into lots of sequential function calls.



Checkpoint

- 1** (*True/False*) The following declaration creates a seven-element `vector<int>`:

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

Answer: False. Actually, the declaration creates a one-element `vector<int>` containing the value 7, not a seven-element vector.

- 2** (*True/False*) When a `try` block throws an exception, this immediately terminates the `try` block, skipping any remaining statements in that block. Then, the `catch` block begins executing. If you declared any variables in the `try` block, they remain in scope and are accessible to the `catch` block.

Answer: False. Actually, variables in the `try` block go out of scope and are not accessible to the `catch` block.

- 3** (*True/False*) A catch block should declare its exception parameter as a `const` 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.

Answer: True.

6.16 Wrap-Up

This chapter began our introduction to data structures, using the C++ standard library class templates `array` and `vector` to store data in and retrieve data from lists and tables of values. We 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. You learned how to use the range-based `for` statement to manipulate every element of an `array` or a `vector`. We also demonstrated C++ standard library functions `sort` to sort an unsorted `array` and `binary_search` to search a sorted `array`.

You learned how to declare and manipulate two-dimensional 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 ranges and views to compose lazy pipelines of operations.

In this chapter's Objects-Natural case study, we demonstrated the capabilities of the C++ standard library class template `vector`. We discussed accessing `array` and `vector` elements with bounds checking and demonstrated basic exception-handling concepts. In later chapters, we'll continue our coverage of data structures.

Chapter 7 presents one of C++'s most powerful features—the pointer. Pointers keep track of where data is stored in memory and enable us to manipulate those items in interesting ways. As you'll see, C++ also provides a language element called an `array` (different from class template `array`) that's closely related to pointers. In contemporary C++ code, it's considered better practice to use the `array` class template rather than traditional arrays.

Exercises

- 6.1** (*Fill in the Blanks*) Fill in the blanks in each of the following:

- a) The names of the `array p`'s four elements are _____, _____, _____ and _____.
- b) Naming an `array`, stating its type and specifying the number of elements in the `array` is called _____ the `array`.
- c) When accessing an `array` element, by convention, the first subscript in a two-dimensional `array` identifies an element's _____ and the second subscript identifies an element's _____.
- d) An m -by- n `array` contains _____ rows, _____ columns and _____ elements.
- e) The name of the element in row 3 and column 5 of `array d` is _____.
- f) _____ simplifies your code by enabling the compiler to infer (i.e., determine) an `array`'s element type from the values in its initializer list.
- g) `A(n)` _____ processes a collection of values into a single value—this is one of the key operations in functional-style programming.

- h) `A(n)` _____ receives another function as an argument.
- i) A C++20 _____ is a collection of elements that you can iterate over.
- j) The common functional-style programming operation known as _____ selects only those elements that match a condition.
- k) The common functional-style programming operation known as _____ transforms a range's values to new values, possibly of different types.

6.2 (*True or False*) Determine whether each of the following is *true* or *false*. If *false*, explain why.

- a) To refer to a particular array element, we specify the array's name and element's value.
- b) An array definition reserves space for an array.
- c) To reserve 100 locations for integer array `p`, you write

`p[100];`

- d) The only way to initialize all the elements of a 15-element array to zero is to use a loop that assigns zero to every element.
- e) Functional-style programming emphasizes mutability, commonly performing operations that modify variables' values.
- f) The accumulate algorithm can be used to perform a reduction that totals the elements of an array.
- g) 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.
- h) A `view::range` generates a range of integers from its first argument up to, but not including, its second argument.

6.3 (*Write C++ Statements*) Write C++ statements to accomplish each of the following:

- a) Display the value of element 6 of the character array `alphabet`.
- b) Input a value into element 4 of one-dimensional floating-point array `grades`.
- c) Initialize each of the 5 elements of one-dimensional integer array `values` to 8.
- d) Total and display the elements of array `temperatures` of 100 double elements.
- e) Determine and display the smallest and largest values in array `values` containing 99 double elements.

6.4 (*One-Dimensional array Questions*) Write statements that perform the following one-dimensional array operations:

- a) Initialize the 10 elements of integer array `counts` to zero.
- b) Add 1 to each of the 15 elements of integer array `bonus`.
- c) Read 12 values for the array of doubles named `monthlyTemperatures` from the keyboard.
- d) Print the 5 values of integer array `bestScores`.

6.5 (*Two-Dimensional array Questions*) Consider a 2-by-3 integer array `t`.

- a) Write a declaration for `t`.
- b) How many rows does `t` have?
- c) How many columns does `t` have?
- d) How many elements does `t` have?
- e) Write the names of all the elements in row 1 of `t`.

- f) Write the names of all the elements in column 2 of `t`.
- g) Write a statement that sets the element of `t` in row 0 and column 1 to zero.
- h) Write a series of statements that initialize each element of `t` to zero. Do not use a loop.
- i) Write a nested counter-controlled `for` statement that initializes each element of `t` to zero.
- j) Write a nested range-based `for` statement that initializes each element of `t` to zero.
- k) Write a statement that inputs the values for the elements of `t` from the keyboard.
- l) Write a series of statements that determine and display the smallest value in array `t`.
- m) Write a statement that displays the elements in row 0 of `t`.
- n) Write a statement that totals the elements in column 2 of `t`.
- o) Write a series of statements that prints the array `t` in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

6.6 (*Salesperson Salary Ranges*) Use a one-dimensional array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople each receive \$200 per week plus 9 percent of their weekly gross sales. For example, a salesperson who grosses \$5000 in sales in a week receives \$200 plus 9 percent of \$5000, or a total of \$650. Write a program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

- a) \$200–299
- b) \$300–399
- c) \$400–499
- d) \$500–599
- e) \$600–699
- f) \$700–799
- g) \$800–899
- h) \$900–999
- i) \$1000 and over

6.7 (*Duplicate Elimination with array*) Use a one-dimensional array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, validate it and store it in the array only if it isn't a duplicate of a number already read. After reading all the values, display only the unique values that the user entered. Provide for the “worst case,” in which all 20 numbers are unique. Use the smallest possible array to solve this problem.

6.8 (*Duplicate Elimination with vector*) Reimplement Exercise 6.7 using a `vector`. Begin with an empty `vector` and use its `push_back` function to add each unique value to the `vector`.

6.9 (*Two-Dimensional array Initialization*) Label the elements of a 3-by-5 two-dimensional array `sales` to indicate the order in which they're set to zero by the following program segment:

```

for (size_t row{0}; row < sales.size(); ++row) {
    for (size_t column{0}; column < sales[row].size(); ++column) {
        sales[row][column] = 0;
    }
}

```

6.10 (*Random Sentences*) Write a program that uses random-number generation to compose sentences. Use four arrays of strings called `article`, `noun`, `verb` and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, concatenate it to the previous words in the sentence. Spaces should separate the words. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate and display 20 sentences.



6.11 (*Intro to Data Science: Coin Tossing*) Modify the die-rolling simulation in Section 6.10 to simulate the flipping a coin. Use randomly generated 1s and 2s to represent heads and tails, respectively. Initially, do not include the frequencies and percentages above the bars. Then modify your code to include the frequencies and percentages. Run simulations for 200, 20,000 and 200,000 coin flips. Do you get approximately 50% heads and 50% tails? Do you see the “law of large numbers” in operation here?



6.12 (*Intro to Data Science: Rolling Two Dice*) Write a program that simulates the rolling of two dice. The sum of the two values should then be calculated. [Note: Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] The following diagram shows the 36 possible combinations of the two dice:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

If you roll the dice 36,000,000 times:

- The values 2 and 12 each occur 1/36th (2.778%) of the time, so you should expect about 1,000,000 of each.
- The values 3 and 11 each occur 2/36ths (5.556%) of the time, so you should expect about 2,000,000 of each, and so on.

Roll the two dice 36,000,000 times. Use a one-dimensional array to summarize the frequencies of each possible sum. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7).

6.13 (*What Does This Code Do?*) What does the following program do?

```

1 // ex06_13.cpp
2 // What does this program do?
3 #include <iostream>
4 #include <array>
5
6 constexpr size_t arraySize{10};
7 int whatIsThis(const std::array<int, arraySize>&, size_t); // prototype
8
9 int main() {
10     std::array a{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12     int result{whatIsThis(a, arraySize)};
13
14     std::cout << "Result is " << result << '\n';
15 }
16
17 // What does this function do?
18 int whatIsThis(const std::array<int, arraySize>& b, size_t size) {
19     if (size == 1) { // base case
20         return b[0];
21     }
22     else { // recursive step
23         return b[size - 1] + whatIsThis(b, size - 1);
24     }
25 }
```

6.14 (*Intro to Data Science Challenge: Analyzing the Dice Game Craps*) In this exercise, you'll modify the program of Fig. 5.5 that simulates the dice game craps. The program should play 1000 games and use two arrays to track the total number of games won and lost on the first roll, second roll, third roll, etc. Summarize the results as follows:

- Display a table indicating how many games are won and how many are lost on the first roll, second roll, third roll, etc. Since the game could continue indefinitely, track wins and losses through the first dozen rolls (of a pair of dice), then maintain two counters that keep track of wins and losses after 12 rolls—no matter how long the game gets.
- What are the chances of winning at craps? [Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]
- What is the mean for the length of a game of craps? The median? The mode?
- Do the chances of winning improve with the length of the game?



6.15 (*Converting vector Example of Section 6.15 to array*) Convert the vector example of Fig. 6.14 to use arrays. Eliminate any vector-only features.

6.16 (*What Does This Code Do?*) What does the following program do?

```

1 // ex06_16.cpp
2 // What does this program do?
3 #include <iostream>
4 #include <array>
5
```

(continued...)

```

6  constexpr size_t arraySize{10};
7  void someFunction(const std::array<int, arraySize>&, size_t); // prototype
8
9  int main() {
10    std::array a{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12    std::cout << "The values in the array are:\n";
13    someFunction(a, 0);
14    std::cout << '\n';
15  }
16
17 // What does this function do?
18 void someFunction(const std::array<int, arraySize>& b, size_t current) {
19   if (current < b.size()) {
20     someFunction(b, current + 1);
21     std::cout << b[current] << " ";
22   }
23 }
```

6.17 (*Sales Summary*) Use a two-dimensional array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains:

- a) the salesperson number,
- b) the product number and
- c) the total monetary value of that product sold that day.

Thus, each salesperson submits 0 to 5 sales slips per day. Assume that the information from all the slips for last month is available. Write a program that will read all this information for last month's sales (one salesperson's data at a time) and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, print the results in tabular format with each column representing a particular salesperson and each row representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

6.18 (*Creating Three-Letter Strings from a Five-Letter Word*) Write a program that reads a five-letter word from the user and produces every possible three-letter string, based on the word's letters. For example, the three-letter words produced from the word "bathe" include "ate," "bat," "bet," "tab," "hat," "the" and "tea."

6.19 (*Telephone-Number Word Generator*) Standard telephone keypads contain the digits 0 through 9. The numbers 2 through 9 each have three letters associated with them, as is indicated by the following table:

Digit	Letter	Digit	Letter
2	A B C	6	M N O
3	D E F	7	P Q R S
4	G H I	8	T U V
5	J K L	9	W X Y Z

Fig. 6.15 | Telephone digit-to-letter mappings.

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word “NUMBERS.”

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then no doubt the business will receive a few more calls. Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., “TAKEOUT”). Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It’s possible, however, that the owner of a barber shop would be pleased to know that the shop’s telephone number, 424-7288, corresponds to “HAIRCUT.” A veterinarian with the phone number 738-2273 would be happy to know that the number corresponds to “PETCARE.”

Write a program that, given a seven-digit number, displays every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

6.20 (Functional-Style Programming: Cubing Integers) Use the techniques from Section 6.14 to calculate the cubes of the numbers 1–5.

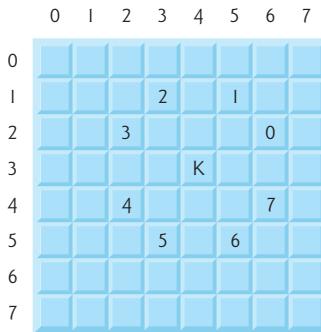
6.21 (Functional-Style Programming: Summing the Triples of the Even Integers from 2 through 10) Use the techniques from Section 6.14 to calculate the total of the triples of the even integers from 2 through 10.

6.22 (Functional-Style Programming: Filtering and Mapping with an array) Create an array called `numbers` containing 1 through 15, then perform the following tasks and display the results:

- Filter the elements of `numbers`, keeping only the even elements.
- Map the elements of `numbers` to their squares.
- Filter the elements of `numbers`, keeping only the even elements, then map them to their squares.

6.23 (Knight’s Tour: Intro to AI with Heuristic Programming) One of the more interesting puzzlers for chess buffs is the Knight’s Tour problem. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth in this exercise.

The knight makes L-shaped moves—two squares in one direction, then one in a perpendicular direction. Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown below:



- Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a 1 in the first square you move to, a 2 in the second square, a 3 in the third, etc. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?
- Now let's develop a program that will move the knight around a chessboard. The board is represented by an 8-by-8 two-dimensional array named `board` with each square initialized to zero. We describe each of the eight possible moves in terms of their horizontal and vertical components. For example, a move of type 0, as shown in the preceding diagram, consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

```

horizontal[0] = 2      vertical[0] = -1
horizontal[1] = 1      vertical[1] = -2
horizontal[2] = -1     vertical[2] = -2
horizontal[3] = -2     vertical[3] = -1
horizontal[4] = -2     vertical[4] = 1
horizontal[5] = -1     vertical[5] = 2
horizontal[6] = 1      vertical[6] = 2
horizontal[7] = 2      vertical[7] = 1

```

Let the variables `currentRow` and `currentColumn` indicate the row and column of the knight's current position. To make a move of type `moveNumber`, where `moveNumber` is between 0 and 7, your program uses the statements

```

currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];

```

Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square and, of course, test every potential move to ensure that the knight does not land off the chessboard. Now write a pro-

gram to move the knight around the chessboard. Run the program. How many moves did the knight make?

- c) After attempting to write and run a Knight's Tour program, you've probably developed some valuable insights. We'll use these to develop a **heuristic** (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic can significantly improve the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome (or inaccessible) squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so when the board gets congested near the end of the tour, there will be a greater chance of success.

We may develop an "accessibility heuristic" by classifying each square according to how accessible it is, then always moving the knight to the least-accessible square (using the knight's L-shaped moves, of course). We label a two-dimensional array **accessibility** with numbers indicating how many squares can access a particular square is accessible. On a blank chessboard, each center square is rated as 8, each corner square is rated as 2, and the other squares have accessibility numbers of 3, 4 or 6 as follows:

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Now write a Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, choose any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

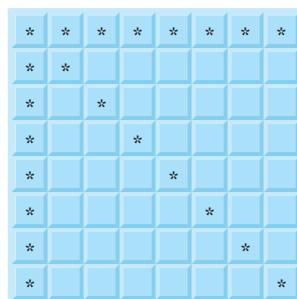
- d) Write a version of the Knight's Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square where the next move would arrive at a square with the lowest accessibility number.

6.24 (Knight's Tour: Brute Force Approaches) In Exercise 6.23, we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue increasing in power, we'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. This is the "brute force" approach to problem-solving.

- Use random number generation to enable the knight to walk around the chessboard at random in its legitimate L-shaped moves. Your program should run one tour and print the final chessboard. How far did the knight get?
- The preceding program likely produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in a neat tabular format. What was the best result?
- The preceding program likely gave you some "respectable" tours but no full tours. Now, let your program run until it produces a full tour. [Caution: This could run for hours on a powerful computer.] Track the number of tours of each length, and print a table of results when the program encounters the first full tour. How many tours did your program attempt before producing a full tour? How much time did it take?
- Compare the brute force version of the Knight's Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute-force problem-solving in general.

6.25 (Eight Queens) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other, i.e., no two queens are in the same row, the same column, or along the same diagonal? Use the thinking developed in Exercise 6.23 to formulate a heuristic for solving the Eight Queens problem. Run your program. [Hint: It's possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as in the following diagram:



Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?]

6.26 (Eight Queens: Brute Force Approaches) In this exercise, you'll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 6.25.

- Solve the Eight Queens exercise using the random brute force technique developed in Exercise 6.24.
- Use an exhaustive technique, i.e., try all possible combinations of eight queens.
- Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight's Tour problem?
- Compare and contrast the random and exhaustive brute force approaches in general.

6.27 (Knight's Tour: Closed-Tour Test) In the Knight's Tour, a full tour occurs when the knight makes 64 moves, touching each square of the board once and only once. A closed tour occurs when the 64th move is one move away from the location where the knight started the tour. Modify the Knight's Tour program you wrote in Exercise 6.23 to test for a closed tour if a full tour has occurred.

6.28 (The Sieve of Eratosthenes) A prime integer is any integer evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

- Create a `bool` array with all elements initialized to `true`. The array elements with prime subscripts will remain `true`. All other array elements will eventually be set to `false`. You'll ignore elements 0 and 1 in this exercise.
- Starting with array subscript 2, every time an array element is found whose value is `true`, loop through the remainder of the array and set to `false` every element whose subscript is a multiple of the subscript for the element with value `true`. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to `false` (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to `false` (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to `true` indicate that the subscript is a prime number. These element's indexes can then be printed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 2 and 999.

Recursion Exercises

6.29 (Palindromes) A palindrome is a string that's spelled the same way forward and backward. Examples of palindromes include "radar" and "able was i ere i saw elba." Write a recursive function `testPalindrome` that returns `true` if a `string` is a palindrome, and `false` otherwise. Note that like an `array`, the square brackets (`[]`) operator can be used to iterate through the characters in a `string`.

6.30 (Eight Queens) Modify the Eight Queens program you created in Exercise 6.25 to solve the problem recursively.

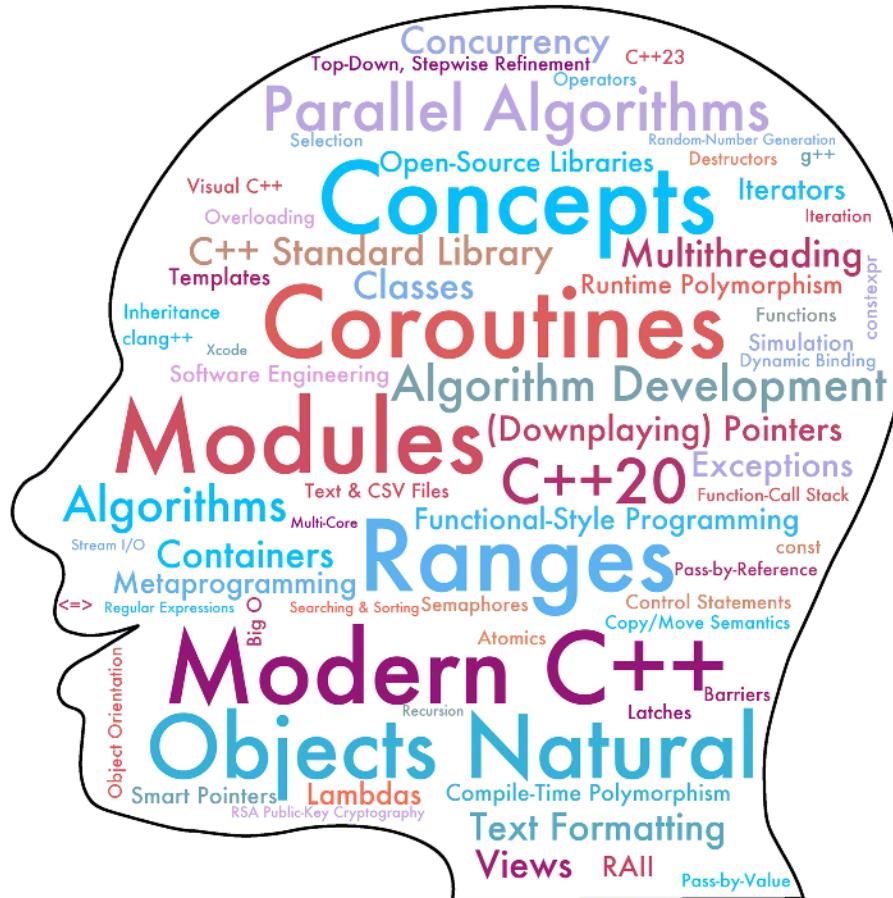
6.31 (Print a vector) Write a recursive function `printVector` that takes a `vector` of `ints`, a starting subscript and an ending subscript as arguments, returns nothing and prints the `vector`. The function should stop processing and return when the starting subscript equals the ending subscript.

6.32 (*Print a String Backward*) Write a recursive function `stringReverse` that takes a `string` and a starting subscript as arguments, prints the string backward and returns nothing. The function should stop processing and return when the end of the string is encountered. Note that like an array the square brackets `([])` operator can be used to iterate through the characters in a `string`.

6.33 (*Find the Minimum Value in an array*) Write a recursive function `recursiveMinimum` that takes an integer array, a starting subscript and an ending subscript as arguments, and returns the smallest element of the array. The function should stop processing and return when the starting subscript equals the ending subscript.

7

(Downplaying) Pointers in Modern C++



Objectives

In this chapter, you'll:

- Learn what pointers are.
- Declare and initialize pointers.
- Use the pointer operators & (address) and * (indirection).
- Compare the capabilities of pointers and references.
- Pass arguments to functions by reference using pointers.
- Use pointer-based arrays and strings mostly in legacy code.
- Use `const` with pointers and the data they point to.
- Determine the number of bytes that store a value of a particular type using `sizeof`.
- Understand legacy-code pointer expressions and pointer arithmetic.
- Use `nullptr` to represent pointers to nothing.
- Use functions `begin` and `end` with pointer-based arrays.
- Learn 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 `spans` to create views into built-in arrays, `std::arrays` and `std::vectors` and to process built-in arrays safely..

Outline

7.1	Introduction	7.7.2	Using a Nonconstant Pointer to Constant Data
7.2	Pointer Variable Declarations and Initialization	7.7.3	Using a Constant Pointer to Nonconstant Data
7.2.1	Declaring Pointers	7.7.4	Using a Constant Pointer to Constant Data
7.2.2	Initializing Pointers	7.8	<code>sizeof</code> Operator
7.2.3	Null Pointers	7.9	Pointer Expressions and Pointer Arithmetic
7.3	Pointer Operators	7.9.1	Adding Integers to and Subtracting Integers from Pointers
7.3.1	Address (<code>&</code>) Operator	7.9.2	Subtracting One Pointer from Another
7.3.2	Indirection (<code>*</code>) Operator	7.9.3	Pointer Assignment
7.3.3	Using the Address (<code>&</code>) and Indirection (<code>*</code>) Operators	7.9.4	Cannot Dereference a <code>void*</code> Pointer
7.4	Pass-by-Reference with Pointers	7.9.5	Comparing Pointers
7.5	Built-In Arrays	7.10	Objects-Natural Case Study: C++20 <code>spans</code> —Views of Contiguous Container Elements
7.5.1	Declaring and Accessing a Built-In Array	7.11	A Brief Intro to Pointer-Based Strings
7.5.2	Initializing Built-In Arrays	7.11.1	Command-Line Arguments
7.5.3	Passing Built-In Arrays to Functions	7.11.2	Revisiting C++20's <code>to_array</code> Function
7.5.4	Declaring Built-In Array Parameters	7.12	Looking Ahead to Other Pointer Topics
7.5.5	Standard Library Functions <code>begin</code> and <code>end</code>	7.13	Wrap-Up Exercises
7.5.6	Built-In Array Limitations		
7.6	Using C++20 <code>to_array</code> to convert a Built-in Array to a <code>std::array</code>		
7.7	Using <code>const</code> with Pointers and the Data Pointed To		
7.7.1	Using a Nonconstant Pointer to Nonconstant Data		

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 or “smart pointer” objects (Section 11.5) over pointers,
- using `std::array` and `std::vector` objects (Chapter 6) over built-in pointer-based arrays, and
- using `std::string` objects (Chapters 2 and 8) and `std::string_view` objects (Section 8.11) over pointer-based C-strings.

Sometimes Pointers Are Still Required

You'll frequently encounter pointers, pointer-based arrays (also called C-style arrays) and pointer-based C-strings 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 program-

mers will use the C++ standard library's existing dynamic containers like `vector` and the containers we discuss in Chapter 13,

- process command-line arguments, which a program receives as a pointer-based array of C-strings, and
- pass arguments by reference if there's a possibility of a `nullptr`¹ (i.e., a pointer to nothing; Section 7.2.2)—a reference must refer to an actual object.²

Pointer-Related C++ Core Guidelines

We mention C++ Core Guidelines for avoiding pointers to make your code safer and more robust, pointer-based arrays and pointer-based strings. For example, several guidelines recommend implementing pass-by-reference using references rather than pointers.³



C++20 Features for Avoiding Pointers

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`. In this chapter's Objects-Natural case study (Section 7.10), you'll see that `spans` 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. We introduce the `sizeof` operator to determine the number of bytes that store values of particular fundamental types and pointers. We also 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 C-strings and pointer-based arrays.

-
1. C++ Core Guidelines, "F.60: Prefer T* over T& When "No Argument" Is a Valid Option." Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-ptr-ref>.
 2. In modern C++, you can represent the presence or absence of an object with `std::optional` (<https://en.cppreference.com/w/cpp/utility/optional>).
 3. C++ Core Guidelines, "F: Functions." Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-functions>.



Checkpoint

1 (*True/False*) Pointers are powerful and easy to work with.

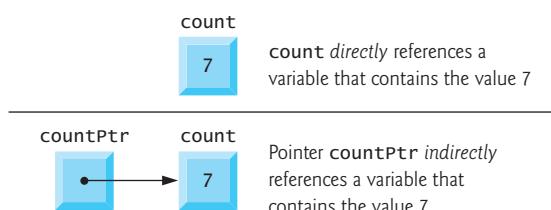
Answer: False. Actually, pointers are powerful but challenging to work with and error-prone.

2 (*True/False*) Generally, you should prefer using `std::array` and `std::vector` objects over built-in pointer-based arrays, and using `std::string` objects and `std::string_view` objects over pointer-based C-strings.

Answer: True.

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 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) as “`countPtr` is a pointer to an `int`”:

```
int* countPtr{nullptr};
```

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

Initialize each pointer with `nullptr` or a memory address. A **null pointer** has the value `nullptr` and “points to nothing.” 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

In legacy C++, null pointers were specified by 0 or `NULL`. `NULL` is defined in several standard library headers to represent the value 0. Initializing a pointer with `NULL` is equivalent to initializing it to 0. For modern C++, the C++ Core Guidelines recommend always using `nullptr` because it’s more readable and cannot be confused with an `int` value.⁴

4. C++ Core Guidelines, “ES.47: Use `nullptr` Rather Than 0 or `NULL`.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nullptr>.



Sec



CG



Checkpoint

1 *(Fill-in)* A variable name directly references a value, and a pointer _____ references a value.

Answer: indirectly.

2 *(Fill-in)* Modern C++ uses _____ to represent a null pointer rather than 0 or NULL.
Answer: nullptr.

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

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

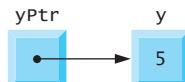
the following statement assigns the variable y's address to the pointer variable yPtr:

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

Variable yPtr is said to “point to” y. Now, yPtr indirectly references the variable y's value.

The & in the preceding statement is not a reference variable declaration—in that context, & would be preceded by a type name and 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 representing the pointer yPtr in memory to the box representing 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 expressions that result in temporary values (like calculations results).



Checkpoint

1 *(Code)* Assume the int variable grade exists. Write a statement that declares the pointer gradePtr and initializes it with the variable grade's address.

Answer: `int* gradePtr=&grade;`

- 2** (*True/False*) In a reference variable declaration, & is preceded by a type name and the & is part of the type.

Answer: True.

- 3** (*Fill-in*) The address operator's operand must be a(n) _____ —the operator cannot be applied to literals or expressions that result in temporary values (like calculations results).

Answer: *lvalue*.

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 known as the **indirection** or **dereferencing operator**. If *yPtr* points to *y* and *y* contains 5 (as in the preceding diagrams), then the following statement displays *y*'s value (5):

```
std::cout << *yPtr << '\n';
```

as would:

```
std::cout << y << '\n';
```

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 *lvalue* referring to *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 Behavior

- Err** Dereferencing an uninitialized pointer or a pointer that no longer points to an object results in undefined behavior that could cause a fatal runtime error. This also could lead to accidentally modifying data, allowing the program to run to completion, possibly with incorrect results. An attacker might exploit this potential security flaw to access data, overwrite data or even execute malicious code.^{5,6,7} Using the result of dereferencing a null pointer or a pointer that no longer points to an object is undefined behavior that often causes a fatal execution-time error. If you must use pointers, ensure that a pointer is not `nullptr` before dereferencing it.
- Sec**

5. “Undefined Behavior.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Undefined_behavior.

6. “Common Weakness Enumeration.” CWE. Accessed April 14, 2023. <https://cwe.mitre.org/data/definitions/824.html>.

7. “Dangling Pointer.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Dangling_pointer.



Checkpoint

- 1** (*True/False*) If `countPtr` points to `count` and `count` contains 17, the following statement displays `count`'s value (17):

```
std::cout << countPtr* << '\n';
```

Answer: False. `countPtr*` should be `*countPtr`.

- 2** (*True/False*) The compiler ignores operations that dereference an uninitialized pointer or a pointer that no longer points to an object.

Answer: False. Actually, these operations are undefined behaviors that could cause fatal runtime errors.

- 3** (*True/False*) If you must use pointers, ensure that a pointer is not `nullptr` before dereferencing it.

Answer: True.

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

Figure 7.1 demonstrates the `&` and `*` pointer operators, which have the third-highest precedence. See https://en.cppreference.com/w/cpp/language/operator_precedence for the complete operator-precedence chart. Memory locations are output by `<<` in this example as hexadecimal (i.e., base-16) integers. (See the Number Systems appendix at <https://deitel.com/cphptp11> for more information on hexadecimal integers.) The output shows that variable `a`'s address (line 9) and `aPtr`'s value (line 10) are identical, confirming that `a`'s address was indeed assigned to `aPtr` (line 7). The outputs from lines 11–12 confirm that `*aPtr` has the same value as `a`. **The displayed memory addresses are compiler- and platform-dependent.** They typically change with each program execution, so you'll likely see different addresses.

```

1 // fig07_01.cpp
2 // Pointer operators & and *.
3 #include <iostream>
4
5 int main() {
6     constexpr int a{7}; // initialize a with 7
7     const int* aPtr{&a}; // initialize aPtr with address of int variable a
8
9     std::cout << "The address of a is " << &a
10    << "\nThe value of aPtr is " << aPtr;
11    std::cout << "\n\nThe value of a is " << a
12    << "\nThe value of *aPtr is " << *aPtr << '\n';
13 }
```

The address of a is 000000D649EFFD00
 The value of aPtr is 000000D649EFFD00

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 and
- **pass-by-reference with a pointer argument** (sometimes called pass-by-pointer).

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 pass large data objects by reference to avoid the overhead of copying objects. 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 cube an integer. Figure 7.2 passes variable `number` by value (line 12) to function `cubeByValue` (lines 17–19), which **cubes the local copy of its argument** and passes the result back to `main` using a `return` statement (line 18).⁸ We store the new value in `number` (line 12), overwriting its original value.

```

1 // fig07_02.cpp
2 // Pass-by-value used to cube a variable's value.
3 #include <format>
4 #include <iostream>
5
6 int cubeByValue(int n); // prototype
7
8 int main() {
9     int number{5};
10
11     std::cout << std::format("Original value of number is {}\n", number);
12     number = cubeByValue(number); // pass number by value to cubeByValue
13     std::cout << std::format("New value of number is {}\n", number);
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 }
```

```
Original value of number is 5
New value of number is 125
```

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

An Example of Pass-By-Reference with Pointers

Figure 7.3 passes the variable `number` to `cubeByReference` using pass-by-reference with a pointer argument (line 13). So, the address of `number` is passed to the function. Function

8. We also could calculate the cube of `n` with `std::pow(n, 3)`.

`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:

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube *nPtr
```

```

1 // fig07_03.cpp
2 // Pass-by-reference with a pointer argument used to cube a
3 // variable's value.
4 #include <iostream>
5 #include <iomanip>
6
7 void cubeByReference(int* nPtr); // prototype
8
9 int main() {
10     int number{5};
11
12     std::cout << std::format("Original value of number is {}\n", number);
13     cubeByReference(&number); // pass number address to cubeByReference
14     std::cout << std::format("New value of number is {}\n", number);
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 }
```

```
Original value of number is 5
New value of number is 125
```

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

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. It does not need to return a value because line 19 assigns the cubed value directly to `number` in `main`.

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. Instead, a pointer to that variable is passed by value. That pointer's value is copied into the function's corresponding pointer parameter. The called function can then dereference the pointer parameter to access the caller's variable, thus accomplishing pass-by-reference.

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

Figures 7.4 and 7.5 graphically analyze the execution of Figs. 7.2 and 7.3, respectively. The rectangle above a given expression or variable contains the value 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() {  
    int number{5};  
    number = cubeByValue(number);  
}
```

Step 2: After cubeByValue receives the call:

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

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

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

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

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

Step 5: After main completes the assignment to number:

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

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

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

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

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

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

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

Step 5: After cubeByReference returns to main:

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

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



Checkpoint

- 1** (*Fill-in*) There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with a reference argument and _____.

Answer: pass-by-reference with a pointer argument (sometimes called pass-by-pointer)

- 2** (*True/False*) Pointers, like references, can be used to modify variables in the caller or pass large data objects by reference to avoid the overhead of copying objects.

Answer: True.

- 3** (*True/False*) When calling a function that receives a pointer, pass a variable's address by applying the * operator to the variable's name.

Answer: False. Actually, when calling a function that receives a pointer, pass a variable's address by applying the & (address operator) to the variable's name.

7.5 Built-In Arrays

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



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 (Section 7.6) to convert them to `std::arrays`, or
- process them more safely using C++20's `spans` (Section 7.10).

Sometimes built-in arrays are required, such as when processing command-line arguments (Section 7.11).



Checkpoint

- 1** (*True/False*) New applications generally should use `std::array` and `std::vector` to create safer, more robust applications than is possible with built-in arrays.

Answer: True.

- 2** (*True/False*) Built-in arrays, `std::array` objects and `std::vector` objects always know their own size, even when passed to other functions.

Answer: False. Actually, only `std::array` and `std::vector` objects know their own size, even when passed to other functions.

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

```
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, `std::array`'s at member function does bounds checking. Built-in arrays do not have a range-checked at indexing function, but you can use the Guidelines Support Library (GSL) function `gsl::at`, which receives as arguments the array and an index. This function's first argument must be the array's name, not a pointer to the array's first element.



Checkpoint

- 1 (*Code*) Write code to reserve seven elements for a built-in array of `ints` named `counts`,

Answer: `int counts[7];`

- 2 (*True/False*) The subscript `([])` operator does not provide bounds checking for `std::arrays` or built-in arrays.

Answer: True.

- 3 (*Fill-in*) `std::array`'s _____ does bounds checking.

Answer: at member function.

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 `ints`. If you provide fewer initializers than the number of elements, the remaining elements are **value initialized** by default:

- fundamental numeric types are set to 0,
- `bools` are set to `false`,
- pointers are set to `nullptr`, and
- objects receive the default initialization specified by their class definitions (which we'll discuss in Chapter 9).

If you provide too many initializers, a compilation error occurs.

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

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



Checkpoint

- 1 (*Code*) Write code to initialize the four elements of built-in `double` array `temperatures` to 97.4, 99.1, 98.3 and 98.6.

Answer: `double temperatures[]{97.4, 99.1, 98.3, 98.6};`

- 2 (*True/False*) The following code will generate a compilation error:

```
int m[]{50, 20, 30};
```

Answer: False. Even though the square brackets are empty, the compiler will infer that the built-in array `m`'s size is 3, based on the number of initializers.

7.5.3 Passing Built-In Arrays to Functions

The value of a built-in array's name is implicitly convertible to a pointer to the array's first element. This is known as **decaying to a pointer**. Given the array

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

the name `n` is implicitly convertible to `&n[0]`, which is a pointer to the element containing 50. You don't need to take a built-in array's address (`&`) to pass it to a function—just 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 of the array's elements in the caller—unless the parameter is declared `const` to prevent the argument array in the caller from being modified (discussed in Section 7.7).



Checkpoint

- 1 *(True/False)* You must take a built-in array's address (`&`) to pass it to a function.

Answer: False. You can just pass the array's name, which decays to a pointer to the array's first element.

- 2 *(Fill-in)* A function that receives a pointer to a built-in array variable in the caller can modify that array in the caller unless the parameter is declared _____.

Answer: `const`.

7.5.4 Declaring Built-In Array Parameters

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

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

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

The preceding header also can be written as

```
int sumElements(const int* values, size_t numberofElements)
```

The compiler does not differentiate between a function that receives a pointer and one 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 pointer to a built-in array's first element vs. a single variable being passed by reference.



The C++ Core Guidelines say not to pass built-in arrays to functions as just a pointer.⁹ Instead, 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.



Checkpoint

- 1 *(True/False)* Built-in arrays don't know their own size, so a function that processes a built-in array also should receive its size.

Answer: True.

9. C++ Core Guidelines, "I.13: Do not pass an array as a single pointer." Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array>.

- 2** (*Code*) Rewrite the following function header using pointer notation for the array parameter:

```
double average(const int grades[], size_t numberofStudents)
```

Answer: `double average(const int* grades, size_t numberofStudents)`

- 3** (*True/False*) Passing a span is superior to passing a built-in array and its size to a function.

Answer: True.

7.5.5 Standard Library Functions `begin` and `end`

In Section 6.12, we sorted a `std::array` of strings called `colors` as follows:

```
std::sort(std::begin(colors), std::end(colors));
```

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

```
std::sort(std::begin(n), std::end(n)); // sort built-in array n
```

For a built-in array, `begin` and `end` work only with the array's name, not with a pointer to the array's first element. Again, you should pass built-in arrays to other functions using C++20 spans, which we demonstrate in Section 7.10.



Checkpoint

- 1** (*True/False*) Function `sort` (and many other C++ standard library functions) also can be applied to built-in arrays.

Answer: True.

- 2** (*Code*) Write a line of code to sort built-in array `values` with `std::sort`, using `std::begin` and `std::end`.

Answer: `std::sort(std::begin(values), std::end(values));`

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.
- Their elements cannot be assigned all at once other built-in arrays with simple assignments, as in `builtInArray1 = builtInArray2`.
- They do not know their own size, so a function that processes a built-in array typically requires both the built-in array's name and 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 Using C++20 `to_array` to Convert a Built-in Array to a `std::array`

CG In industry, you'll encounter C++ legacy code containing 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 do not decay to pointers when you pass them to functions.¹⁰ C++20's `std::to_array` function¹¹ (header `<array>`) conveniently creates 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 9–15) 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 element type of the `std::array` over which it iterates.

```

1 // fig07_06.cpp
2 // C++20: Creating std::arrays with to_array.
3 #include <format>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8     // generic lambda to display a collection of items
9     const auto display{
10         [](const auto& items) {
11             for (const auto& item : items) {
12                 std::cout << std::format("{} ", item);
13             }
14         }
15     };
16
17 const int values1[]{10, 20, 30};
18
19 // creating a std::array from a built-in array
20 const auto array1{std::to_array(values1)};
21
22 std::cout << std::format("array1.size() = {}\n", array1.size())
23     << "array1: ";
24 display(array1); // use lambda to display contents
25
26 // creating a std::array from an initializer list
27 const auto array2{std::to_array({1, 2, 3, 4})};
28 std::cout << std::format("\n\narray2.size() = {}\n", array2.size())
29     << "array2: ";
30 display(array2); // use lambda to display contents
31
32 std::cout << '\n';
33 }
```

Fig. 7.6 | C++20: Creating `std::arrays` with `to_array`. (Part 1 of 2.)

10. C++ Core Guidelines, “SL.con.1: Prefer Using STL array or vector Instead of a C array.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-arrays>.
11. Zhihao Yuan, “`to_array` from LFTS with Updates,” July 17, 2019. Accessed April 14, 2023. <https://wg21.link/p0325>.

```
array1.size() = 3  
array1: 10 20 30  
  
array2.size() = 4  
array2: 1 2 3 4
```

Fig. 7.6 | C++20: Creating `std::arrays` with `to_array`. (Part 2 of 2.)

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

Line 20 creates a three-element `std::array` of `ints` by copying the contents of the built-in array `values1`. We use `auto` to infer the `std::array` variable's type and size. A compilation error occurs if we declare the array's type and size explicitly and it does not match `to_array`'s return value. We assign the result to the variable `array1`. Lines 22 and 24 display the `std::array`'s size and contents to confirm that it was created correctly.

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

Line 27 shows that `to_array` can create a `std::array` from an initializer list. Lines 28 and 30 display the array's size and contents to confirm that it was created correctly.



Checkpoint

- 1 *(Fill-in)* The _____ function (header `<array>`) makes it convenient to create a `std::array` from a built-in array or an initializer list.

Answer: `std::to_array`.

- 2 *(True/False)* Rather than built-in arrays, use `std::arrays` and `std::vectors` because they're safer and know their own size when you pass them to functions.

Answer: True.

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

This section discusses combining `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 it can and cannot modify.

There are four ways to declare pointers:

- 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 in¹²

```
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.

```
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
   | ~~~~~^~~~~~
```

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

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 or 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.



12. 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.”

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. If the pointer is a function parameter, it's initialized with the pointer 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`.

```
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
```

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 error messages from Clang in Xcode generated by 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.

```

1 // fig07_09.cpp
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4
5 int main() {
6     int x{5};
7     int y{6};
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    std::cout << *ptr << '\n';
15
16    *ptr = 7; // error: *ptr is const; cannot assign new value
17    ptr = &y; // error: ptr is const; cannot assign new address
18 }

```

Compiler error messages from Clang in Xcode:

```

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
    ~~~~ ^

```

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



Checkpoint

- 1 (*Code*) Write code that declares `temperaturePtr` as a nonconstant pointer to constant `double` data and initialize the pointer to `nullptr`.

Answer: `const double* temperaturePtr{nullptr};`

- 2 (*Code*) Consider the following code:

```

int x{3};
int y{5};
int* const ptr{&x};
*ptr = 7;
ptr = &y;

```

Why is the first assignment allowed but the second assignment disallowed?

Answer: The first assignment is allowed because `*ptr` is not `const`. The second assignment is disallowed because `ptr` is `const`, so it cannot be aimed at another variable.

- 3 (*Code*) What's wrong with the following declaration?

```
const double* const temperaturePtr;
```

Answer: `temperaturePtr` is a constant pointer, so it must be initialized to point to a `double` variable.

7.8 `sizeof` Operator

The compile-time unary operator `sizeof` determines the size in bytes of a built-in array, type, variable or constant *during program compilation*. When applied to a built-in array's name, as in Fig. 7.10¹³ (line 13), `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. Array `numbers` has 20 elements (line 10), so it uses 160 bytes in memory. Applying `sizeof` to a pointer (line 21) returns the size of the pointer in bytes (8 on the system we used).

```

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 <format>
5 #include <iostream>
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    std::cout << std::format("Number of bytes in numbers is {}\n",
13                           sizeof(numbers));
14
15    std::cout << std::format("Number of bytes returned by getSize is {}\n",
16                           getSize(numbers));
17 }
18
19 // return size of ptr
20 size_t getSize(double* ptr) {
21    return sizeof(ptr);
22 }
```

```
Number of bytes in the array is 160
Number of bytes returned by getSize is 8
```

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
 Figure 7.11 uses `sizeof` to calculate the number of bytes used to store various standard data types. The output was produced using the Clang compiler in Xcode. Type sizes are platform-dependent. When we run this program on our Windows system, `long` is 4 bytes and `long long` is 8 bytes, whereas they're both 8 bytes on our Mac. In this example, lines 7–15 implicitly initialize each variable to 0 using an empty initializer list, {}.¹⁴

- 13. 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.
- 14. 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.

```

1 // fig07_11.cpp
2 // sizeof operator used to determine standard data type sizes.
3 #include <format>
4 #include <iostream>
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    std::cout << std::format("sizeof c = {}\tsizeof(char) = {}\n",
19        sizeof c, sizeof(char));
20    std::cout << std::format("sizeof s = {}\tsizeof(short) = {}\n",
21        sizeof s, sizeof(short));
22    std::cout << std::format("sizeof i = {}\tsizeof(int) = {}\n",
23        sizeof i, sizeof(int));
24    std::cout << std::format("sizeof l = {}\tsizeof(long) = {}\n",
25        sizeof l, sizeof(long));
26    std::cout << std::format("sizeof ll = {}\tsizeof(long long) = {}\n",
27        sizeof ll, sizeof(long long));
28    std::cout << std::format("sizeof f = {}\tsizeof(float) = {}\n",
29        sizeof f, sizeof(float));
30    std::cout << std::format("sizeof d = {}\tsizeof(double) = {}\n",
31        sizeof d, sizeof(double));
32    std::cout << std::format("sizeof ld = {}\tsizeof(long double) = {}\n",
33        sizeof ld, sizeof(long double));
34    std::cout << std::format("sizeof array = {}\n", sizeof array);
35    std::cout << std::format("sizeof ptr = {}\n", sizeof ptr);
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, consider using the **fixed-size integer types** from header `<cstdint>`. For the complete list, see:

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

Operator `sizeof` can be applied to any expression or type name. When applied to a variable name or 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 will not be evaluated at runtime.



Checkpoint

1 (*True/False*) Applying `sizeof` to a pointer returns the size in bytes of what the pointer points to.

Answer: False. Actually, applying `sizeof` to a pointer returns the size of the pointer in bytes.

2 (*Fill-in*) 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, consider using the _____ from header `<cstdint>`.

Answer: fixed-size integer types.

3 (*True/False*) `sizeof` is a compile-time operator, so its operand will not be evaluated at runtime.

Answer: True.

7.9 Pointer Expressions and Pointer Arithmetic

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

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).¹⁵ They also say to avoid pointer arithmetic because it's highly error-prone.¹⁶ If you need to process built-in arrays, use C++20 **spans** instead (Section 7.10).

CG

Err

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.

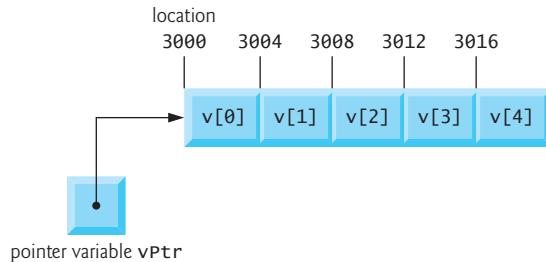
Pointer arithmetic results depend on the size of the memory objects a pointer points to, so pointer arithmetic is machine-dependent.

Most computers today have four-byte (32-bit) or eight-byte (64-bit) integers. Some of the billions of resource-constrained Internet of Things (IoT) devices are built using eight-bit or 16-bit hardware with two-byte integers—the minimum `int` size according to

15. C++ Core Guidelines, “ES.42: Keep Use of Pointers Simple and Straightforward.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr>.

16. C++ Core Guidelines, “Pro.bounds: Bounds Safety Profile.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-bounds>.

the C++ standard. Assume that `int v[5]` exists, that its first element is at memory location 3000 and that ints are stored in four bytes. Also, assume that the 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

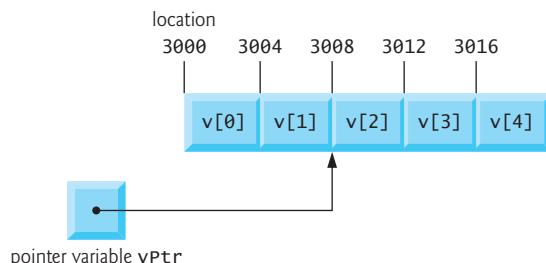
In conventional arithmetic, the addition $3000 + 2$ yields the value 3002 . This is normally not the case with pointer arithmetic:

- Adding an integer to a pointer increments the pointer by that integer times the size of the type to which the pointer refers.
- Subtracting an integer from a pointer decrements the pointer by that integer times the size of the type to which the pointer refers.

The number of bytes depends on the object's data type. For example, the statement

```
vPtr += 2;
```

would produce 3008 ($3000 + 2 * 4$), 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. To increment or decrement a pointer by one element, use increment (`++`) or decrement (`--`) operator, respectively. The statements

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

increment the pointer to point to the built-in array's next element. The statements

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

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

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 expression

```
v2Ptr - vPtr;
```

would determine the number of built-in array elements from `vPtr` to `v2Ptr`—in this case, 2. 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.¹⁷ Also, any pointer to a fundamental type or class type can be assigned to a `void*` (**pointer to 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 `static_cast` to the proper pointer type. For example, to cast a `void*` named `ptr` to an `int*`, you'd use

```
int* intPtr{static_cast<int*>(ptr)};
```

7.9.4 Cannot Dereference a `void*` Pointer

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, contains a memory address for an unknown data type. You cannot use a `void*` in pointer arithmetic, nor can you dereference a `void*` because the compiler does not know the type or size of the data to which the pointer refers.

The allowed operations on `void*` pointers are:

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

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

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

7.9.5 Comparing Pointers

Pointers can be compared using equality and relational operators. Relational operators (`<`, `<=`, `>` and `<=`) compare the addresses stored in the pointers and are meaningful only if the pointers point to elements of the same built-in array. For example, such a comparison might show that one pointer points to a higher-numbered element than the other. A common use of pointer equality comparison is determining whether a pointer has the value `nullptr` (i.e., a pointer to nothing).



Checkpoint

1 *(True/False)* Adding an integer to a pointer increments the pointer by that integer times the size of the type to which the pointer refers.

Answer: True.

2 *(True/False)* There's no bounds checking on pointer arithmetic, so the C++ Core Guidelines recommend using `std::spans` instead. These have bounds checking, which helps you avoid errors.

Answer: True.

3 *(Fill-in)* A pointer of type `void*` cannot be assigned directly to a pointer of another type—the pointer of type `void*` must first be _____ to the proper pointer type.

Answer: `static_cast`.

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 elements but does not have its own copy of those elements.

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 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`¹⁸, which represent both a pointer to the array’s first element and the array’s size. Figure 7.12 demonstrates some key `span` capabilities. We broke this program into parts for discussion purposes.

```

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

```

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

18. C++ Core Guidelines, “R.14: Avoid [] Parameters, Prefer span.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ap>.

Function `displayArray`

Passing a built-in array to a function typically requires the array's name and size. Though the parameter `items` (line 12) is declared with `[]`, it's simply a pointer to an `int`. It 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
- 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 to compare it with passing `spans` in function `displaySpan`, which is the recommended approach.



```

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         std::cout << std::format("{} ", items[i]);
15     }
16 }
17

```

Function `displaySpan`

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



```

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(std::span<const int> items) {
21     for (const auto& item : items) { // spans are iterable
22         std::cout << std::format("{} ", item);
23     }
24 }
25

```

A `span` encapsulates both a pointer and a `size_t` representing the number of elements. When you pass a built-in array (or a `std::array` or `std::vector`) to `displaySpan`, C++ implicitly creates a `span` containing a pointer to the array's first element and its size,

19. C++ Core Guidelines, "ES.42: Keep Use of Pointers Simple and Straightforward." Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr>.

20. C++ Core Guidelines, "I.13: Do Not Pass an Array as a Single Pointer." Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array>.

 which the compiler determines from the array's declaration. This `span` views the data in the original array 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 separately,²¹ as we did in `displayArray`.

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 array's bounds. This fixes the various problems associated with `displayArray` and helps prevent security issues like buffer overflows.

Function `times2`

Because a `span` is a view into an existing container, 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.

```

26 // spans can be used to modify elements in the original data structure
27 void times2(std::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`.

```

33 int main() {
34     int values1[]{1, 2, 3, 4, 5};
35     std::array values2{6, 7, 8, 9, 10};
36     std::vector 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     std::cout << "values1 via displayArray: ";
41     displayArray(values1, 5);
42

```

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

21. C++ Core Guidelines, “F.24: Use a `span<T>` or a `span_p<T>` to Designate a Half-open Sequence.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-range>.

Implicitly Creating spans and Passing Them to Functions

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

```
std::span<const int>
```

so C++ creates a span containing a `const int*` that points to the array's first element and a `size_t` representing the array's size, which the compiler gets from the `values1` declaration (line 34). Because spans can view any contiguous sequence of elements, you may also pass a `std::array` or `std::vector` of ints to `displaySpan` (lines 50 and 52). C++ will create an appropriate span representing a pointer to the container's first element and size. This makes `displaySpan` more flexible than `displayArray`, which could receive only the built-in array in this example.

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

```
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

```
std::span<int>
```

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

```
54 // changing a span's contents modifies the original data
55 times2(values1);
56 std::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
```

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`—the compiler uses CTAD to infer the element type `int` from `values1`'s elements. Lines 61–62 demonstrate the span's `front` and `back` member functions, which return the first and last elements of the view—thus, the first and last elements of `values1`, respectively.

```
59  // spans have various array-and-vector-like capabilities
60  std::span mySpan{values1}; // span<int>
61  std::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
```



A philosophy of the C++ Core Guidelines is to “prefer compile-time checking to runtime checking.”²² This enables the compiler to find and report errors at compile-time, rather than you writing 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 explicitly state the span's type and size, as in

```
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 14.

```
64  // spans can be used with standard library algorithms
65  std::cout << "\n\nSum of mySpan's elements: "
66      << std::accumulate(std::begin(mySpan), std::end(mySpan), 0);
67
```

```
Sum of mySpan's elements: 30
```

Creating Subviews

Sometimes, you might want to process subsets of a span. A span's `first`, `last` and `subspan` member functions create subviews. Lines 70 and 72 use `first` and `last` to get spans representing `values1`'s first three and last three elements, 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 to `displaySpan` to confirm what the subview represents.

22. C++ Core Guidelines, “P.5: Prefer Compile-Time Checking to Run-Time Checking,” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-compile-time>.

```

68 // spans can be used to create subviews of a container
69 std::cout << "\n\nFirst three elements of mySpan: ";
70 displaySpan(mySpan.first(3));
71 std::cout << "\nLast three elements of mySpan: ";
72 displaySpan(mySpan.last(3));
73 std::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 creates a span that views the 3 elements starting from index 1 of `values1`, then passes it to function `times2`. Line 79 displays the updated `values1` elements to confirm the results.

```

76 // changing a subview's contents modifies the original data
77 times2(mySpan.subspan(1, 3));
78 std::cout << "\n\nvalues1 after modifying elements via span: ";
79 displaySpan(values1);
80

```

```
values1 after modifying elements via span: 2 8 12 16 10
```

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, which does not provide range checking. Line 82 displays the element at index 2.²³

```

81 // access a span element via []
82 std::cout << "\n\nThe element at index 2 is: " << mySpan[2];
83 }

```

```
The element at index 2 is: 12
```



Checkpoint

- I *(True/False)* 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 elements but does not have its own copy of those elements.

Answer: True.

23. `std::span` does not provide a range-checked `at` member function, though it might in the future.

- 2** (*Fill-in*) A **span** encapsulates both a pointer and a _____ representing the number of elements.

Answer: `size_t`

- 3** (*True/False*) Because a **span** is a view into an existing container, changing the **span**'s elements changes the container's original data.

Answer: True.

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 pointer-based strings, as inherited from 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.

Sec 

`std::string` is preferred because it eliminates many security problems and bugs caused by manipulating C-strings. However, there are some cases where C-strings are required, such as when processing command-line arguments. Also, if you work with legacy C and C++ programs, you'll likely encounter pointer-based strings.

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 letter z's integer value (122 in the ASCII character set; see the Character Set appendix at <https://deitel.com/cpphtp11>) as type `char`. Similarly, '`\n`' represents the integer value of newline (10 in the ASCII character set).

Pointer-Based Strings

A C-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).

String Literals as Initializers

A string literal may be used as an initializer for a built-in array of `char`s or a variable of type `const char*`. The following declarations each initialize a variable to the string "blue":

```
char color[] {"blue"};
const char* colorPtr {"blue"};
```

The first declaration creates a five-element built-in array `color` containing the characters '`b`', '`l`', '`u`', '`e`' and '`\0`'. The second creates the pointer variable `colorPtr` pointing 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 in which you manually include the terminating '`\0`', as in:

```
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.

When storing a string of characters in a built-in array of `chars`, be sure that it 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

```
std::cout << color;
```

displays the built-in array `color`. The `cout` object 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. Both `cin` and `cout` assume that built-in arrays of `chars` should be processed as strings terminated by null characters. They 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. For example, **command-line arguments** 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 its name when executing it from the command line. 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. 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.

```

1 // fig07_13.cpp
2 // Reading in command-line arguments.
3 #include <format>
4 #include <iostream>
5
6 int main(int argc, char* argv[]) {
7     std::cout << std::format("Number of arguments: {}\n\n", argc);
8
9     for (int i{0}; i < argc; ++i) {
10         std::cout << std::format("{}\n", argv[i]);
11     }
12 }
```

fig07_13 Leila Haufiku 97
Number of arguments: 4

fig07_13
Leila
Haufiku
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 pointer to the first element of a built-in array of `char*`. Some programmers write this as `char** argv`. The array's first element is a C-string for the application name.²⁴ The remaining elements are C-strings for the other command-line arguments.

The command

`fig07_13 Leila Haufiku 97`

passes "Leila", "Haufiku" 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 whitespace, 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 "Leila", "Haufiku" and "97", respectively.

You determine how to use these arguments in your program. Due to the problems we've discussed with C-strings, you should convert the command-line arguments to `std::strings` before using them in your program.

7.11.2 Revisiting C++20's `to_array` Function

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 the `to_array` call.

24. The C++ standard allows implementations to return an empty string for `argv[0]`.

```
1 // fig07_14.cpp
2 // C++20: Creating std::arrays from string literals with to_array.
3 #include <format>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8     // Lambda to display a collection of items
9     const auto display{
10         [](const auto& items) {
11             for (const auto& item : items) {
12                 std::cout << std::format("{} ", item);
13             }
14         }
15     };
16
17     // initializing an array with a string literal
18     // creates a one-element array<const char*>
19     const auto array1{std::array{"abc"}};
20     std::cout << std::format("array1.size() = {}\narray1: ",
21         array1.size());
22     display(array1); // use lambda to display contents
23
24     // creating std::array of characters from a string literal
25     const auto array2{std::to_array("C++20")};
26     std::cout << std::format("\n\narray2.size() = {}\narray2: ",
27         array2.size());
28     display(array2); // use lambda to display contents
29
30     std::cout << '\n';
31 }
```

```
array1.size() = 1
array1: abc

array2.size() = 6
array2: C + + 2 0
```

Fig. 7.14 | C++20: Creating std::arrays from string literals with to_array.

Initializing a std::array from a String Literal Creates a One-Element array
Line 19 creates a one-element array containing a const char* pointing to the C-string "abc".

Passing a String Literal to to_array Creates a std::array of char

On the other hand, passing to_array a string literal (line 25) creates a std::array of chars containing elements for each character and the terminating null character. Lines 25–26 confirm that the array's size is 6. Line 27 confirms the array's contents. The null character does not have a visual representation, so it does not appear in the output.



Checkpoint

1 (*Fill-in*) A C-string is accessed via _____ no matter how long the string is.

Answer: a pointer to its first character.

2 (*True/False*) The following two individual lines of code each create the same C-string:

```
char day[] {"Sunday"};
char day[] {'S', 'u', 'n', 'd', 'a', 'y'};
```

Answer: False. The first built-in char array contains a C-string, including its terminating \0 (null) character. The second contains the individual letters of the word Sunday, but does not have a terminating \0 (null) character so it cannot be treated as a C-string. The following updates the second statement to store a C-string:

```
char day[] {'S', 'u', 'n', 'd', 'a', 'y', '\0'};
```

3 (*True/False*) Not allocating sufficient space in a built-in array of chars to store the null character that terminates a C-string is a logic error. Creating or using a C-string that does not contain a terminating null character can lead to logic errors.

Answer: True.

7.12 Looking Ahead to Other Pointer Topics

In later chapters, we'll introduce additional pointer topics:

- In Chapter 10, OOP: Inheritance and Runtime 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 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, 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 14, Standard Library Algorithms and C++20 Ranges & Views, we show that a function's name can be treated as a pointer to its implementation and that functions can be passed into other functions via function pointers.

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 over pointers, `std::array` and `std::vector` objects to built-in arrays, and `std::string` objects to C-strings.

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. We used the `sizeof` operator to determine the number of bytes used to store values of various types. 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.

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 create `std::arrays` from built-in arrays and C++20’s `spans` as a safer way to process built-in pointer-based arrays. The next chapter discusses typical string-manipulation operations provided by `std::string` and introduces file-processing capabilities.

Exercises

7.1 For each of the following, write C++ statements that perform the specified task. Assume that double-precision, floating-point numbers are stored in eight bytes and that the starting address of the built-in array is at location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

- a) Declare a built-in array of type `double` called `numbers` with 10 elements, and initialize the elements to the values 0.0, 1.1, 2.2, ..., 9.9. Assume that the constant `size` has been defined as 10.
- b) Declare a pointer `nPtr` that points to a variable of type `double`.
- c) Use a `for` statement to display the elements of built-in array `numbers` using array subscript notation. Display each number with one digit to the right of the decimal point.
- d) Write two separate statements that each assign the starting address of built-in array `numbers` to the pointer variable `nPtr`.
- e) Use a `for` statement to display the elements of built-in array `numbers` using pointer/offset notation with pointer `nPtr`.
- f) Use a `for` statement to display the elements of built-in array `numbers` using pointer/offset notation with the built-in array’s name as the pointer.
- g) Use a `for` statement to display the elements of built-in array `numbers` using pointer/subscript notation with pointer `nPtr`.
- h) Refer to element 3 of built-in array `numbers` using array subscript notation, pointer/offset notation with the built-in array’s name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.
- i) Assuming that `nPtr` points to the beginning of built-in array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?
- j) Assuming that `nPtr` points to `numbers[5]`, what address is referenced by `nPtr` after `nPtr -= 4` is executed? What’s the value stored at that location?

7.2 For each of the following, write a statement that performs the specified task. Assume the `double` variables `number1` and `number2` have been declared and `number1` has been initialized to 7.3.

- a) Declare the variable `doublePtr` to be a pointer to an object of type `double` and initialize the pointer to `nullptr`.

- b) Assign the address of the variable `number1` to pointer variable `doublePtr`.
- c) Display the value of the object pointed to by `doublePtr`.
- d) Assign the value of the object pointed to by `doublePtr` to variable `number2`.
- e) Display the value of `number2`.
- f) Display the address of `number1`.
- g) Display the address stored in `doublePtr`. Is the address the same as the address of variable `number1`?

7.3 (Find the Code Errors) Find the error in each of the following program segments. Assume the following declarations and statements:

```
int* zPtr=nullptr; // zPtr will reference built-in array z
int number{0};
int z[5]{1, 2, 3, 4, 5};

a) ++zPtr;
b) // use pointer to get first value of a built-in array
   number = zPtr;
c) // assign built-in array element 2 (the value 3) to number
   number = *zPtr[2];
d) // display entire built-in array z
   for (size_t i{0}; i <= 5; ++i) {
       std::cout << zPtr[i] << '\n';
   }
e) ++z;
```

7.4 (Write C++ Statements) For each of the following, write C++ statements that perform the specified task. Assume that integers are stored in four bytes and that the starting address of the built-in array is at location 1002500 in memory.

- a) Declare an `int` built-in array `values` with five elements initialized to the even integers from 2 to 10. Assume that the constant `size` has been defined as 5.
- b) Declare a pointer `vPtr` that points to an object of type `int`.
- c) Use a `for` statement to display the elements of built-in array `values` using array subscript notation.
- d) Write two separate statements that assign the starting address of built-in array `values` to the pointer variable `vPtr`.
- e) Use a `for` statement to display the elements of built-in array `values` using pointer/offset notation.
- f) Use a `for` statement to display the elements of built-in array `values` using pointer/offset notation with the built-in array's name as the pointer.
- g) Use a `for` statement to display the elements of built-in array `values` using subscript notation and the pointer to the built-in array.
- h) Refer to `values`' element 4 using array subscript notation, pointer/offset notation with the built-in array name's as the pointer, pointer subscript notation and pointer/offset notation.
- i) What address is referenced by `vPtr + 3`? What value is stored at that location?
- j) Assuming `vPtr` points to `values[4]`, what address is referenced by `vPtr -= 4`? What value is stored at that location?

7.5 (Write C++ Statements) For each of the following, write a single statement that performs the specified task. Assume that long variables `value1` and `value2` have been declared, and `value1` has been initialized to 200000.

- Declare the variable `longPtr` as a pointer to an object of type `long`.
- Assign the address of variable `value1` to pointer variable `longPtr`.
- Display the value of the object pointed to by `longPtr`.
- Assign the value of the object pointed to by `longPtr` to variable `value2`.
- Display the value of `value2`.
- Display the address of `value1`.
- Display the address stored in `longPtr`. Is it displayed the same as `value1`'s?

7.6 (Find the Code Errors) Find the error in each of the following segments. If the error can be corrected, explain how.

- `int* number;`
`cout << number << '\n';`
- `double* realPtr{nullptr};`
`long* integerPtr{nullptr};`
`integerPtr = realPtr;`
- `char s[]{"this is a character array"};`
`for (; *s != '\0'; ++s) {`
 `std::cout << *s << ' ';`
`}`
- `double x{19.34};`
`double xPtr{&x};`
`cout << xPtr << '\n';`

7.7 (Cubing the Elements of a span) Use the techniques demonstrated in Fig. 7.12 to create a `cubeElements` function that cubes the elements of its argument. Enable the function to receive as its argument a built-in array, `std::array` or `std::vector`. Test your function using `values1`, `values2` and `values3`:

```
int values1[] {1, 2, 3, 4, 5};
std::array values2{6, 7, 8, 9, 10};
std::vector values3{11, 12, 13, 14, 15};
```

Use Fig. 7.12's `displaySpan` function to display `values1`, `values2` and `values3` before and after cubing each.

7.8 (Simulation: The Tortoise and the Hare) Let's re-create the classic race of the tortoise and the hare (https://en.wikipedia.org/wiki/The_Tortoise_and_the_Hare). You'll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a position along the race course, and the finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally, the contenders lose ground.

There is a clock that ticks once per second. With each clock tick, your program should use functions `moveTortoise` and `moveHare` to adjust the animals' positions according to Fig. 7.15's rules. These functions should use pointer-based pass-by-reference to modify the position of the tortoise and the hare.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 7.15 | Rules for moving the tortoise and the hare.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the “starting gate”). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in Fig. 7.15 by producing a random integer i in the range $1 \leq i \leq 10$. For the tortoise, perform a “fast plod” when $1 \leq i \leq 5$, a “slip” when $6 \leq i \leq 7$ or a “slow plod” when $8 \leq i \leq 10$. Use a similar technique to move the hare.

Begin the race by displaying

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

For each clock tick (i.e., each iteration of a loop), display a 70-position line showing the letter T in the tortoise’s position and H in the hare’s position. When the contenders land on the same square, the tortoise bites the hare, and your program should display "OUCH!!!" beginning at that position. All positions other than the T, the H or the OUCH!!! (in case of a tie) should be blank.

After displaying each line, test whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display "TORTOISE WINS!!! YAY!!!" If the hare wins, display "Hare wins. Yuch." If both animals win on the same clock tick, you may want to favor the tortoise (the “underdog”), or you may want to display It's a tie. If neither animal wins, perform the loop again to simulate the next clock tick.

Special Section—Building Your Own Computer as a Virtual Machine

The next several exercises temporarily diverge from the world of high-level language programming. We “peel open” a fake simple computer and look at its internal structure. We introduce simple machine-language programming for this computer and write several machine-language programs. To make this an especially valuable experience, we then build a software-based *simulation* of this computer on which you actually can execute your machine-language programs! Such a simulated computer is often called a **virtual machine**.

7.9 (Machine-Language Programming) Let’s create a computer we’ll call the Simpletron. As its name implies, it’s a simple machine, but as we’ll soon see, it’s powerful. The

Simpletron runs programs written in the only language it directly understands—**Simpletron Machine Language** or **SML** for short.

The Simpletron contains an **accumulator**—a “special register” in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of **words**. A word is a signed four-digit decimal number, such as +3364, -1293, +0007, -0001, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, ..., 99.

Before running an SML program, we must **load** the program into memory. The first instruction (or statement) of every SML program is always placed in location 00.

Each **SML instruction** occupies one word of the Simpletron’s memory, so instructions are signed four-digit decimal numbers. We assume an SML instruction’s sign is always plus, but a data word’s sign may be plus or minus. Each Simpletron memory location may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. Each SML instruction’s first two digits are the **operation code** specifying the operation to perform. The SML operation codes are summarized in the following table. They should be defined in a scoped **enum** named **OperationCodes**—each constant is shown with the value you should explicitly assign to it:

Operation code	Meaning
<i>Input/output operations:</i>	
<code>read = 10</code>	Read a word from the keyboard into a specific memory location.
<code>write = 11</code>	Write a word from a specific memory location to the screen.
<i>Load/store operations:</i>	
<code>load = 20</code>	Load a word from a specific memory location into the accumulator.
<code>store = 21</code>	Store a word from the accumulator into a specific memory location.
<i>Arithmetic operations:</i>	
<code>add = 30</code>	Add a word from a specific memory location to the word in the accumulator (leave the result in the accumulator).
<code>subtract = 31</code>	Subtract a word from a specific memory location from the word in the accumulator (leave the result in the accumulator).
<code>divide = 32</code>	Divide a word from a specific memory location into the word in the accumulator (leave the result in the accumulator).
<code>multiply = 33</code>	Multiply a word from a specific memory location by the word in the accumulator (leave the result in the accumulator).
<i>Transfer-of-control operations:</i>	
<code>branch = 40</code>	Branch to a specific memory location.
<code>branchNeg = 41</code>	Branch to a specific memory location if the accumulator is negative.
<code>branchZero = 42</code>	Branch to a specific memory location if the accumulator is zero.
<code>halt = 43</code>	Halt—i.e., the program has completed its task.

An SML instruction’s last two digits are the **operand**—the memory location containing the word to which the operation applies.

Sample SML Program That Adds Two Numbers

Let's consider several simple SML programs. The following SML program reads two numbers from the keyboard, then computes and prints their sum:

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

The instruction +1007 reads the first number from the keyboard and places it into location 07. Then +1008 reads the next number into location 08. The *load* instruction, +2007, copies the first number into the accumulator. The *add* instruction, +3008, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, +2109, copies the result from the accumulator into memory location 09, from which the *write* instruction, +1109, then takes the number and prints it as a signed four-digit decimal number to the screen. The *halt* instruction, +4300, terminates execution.

Sample SML Program That Determines the Largest of Two Values

The next SML program reads two numbers from the keyboard, then determines and prints the larger value:

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

The instruction +4107 is a conditional transfer of control, like an `if` statement.

Now write SML programs to accomplish each of the following tasks.

- a) Use a sentinel-controlled loop to read positive integers, then compute and print their sum.
- b) Use a counter-controlled loop to read seven numbers, some positive and some negative. Compute and print their average.
- c) Read a series of numbers. Determine and print the largest number. The first number read indicates how many numbers should be processed.

7.10 (*A Computer Simulator*) It may at first seem outrageous, but in this exercise you'll build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of **software-based simulation** to create a **software model** of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 7.9!

When you run your Simpletron simulator, it should begin by printing:

```
***           Welcome to Simpletron      ***
***                                     ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** Location number and a question mark (?). ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program.                      ***
```

Simulate the memory of the Simpletron with a 100-element one-dimensional array `memory`. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 7.9:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins ***
```

The SML program has now been placed (or loaded) into the array `memory`. Next, the Simpletron executes the SML program. It begins with the instruction in location 00 and continues sequentially unless directed to some other part of the program by a transfer of control.

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to store the number of the memory location (00 to 99) containing the instruction being performed. Use the variable `operationCode` to store the operation currently being performed (the instruction word's left two digits). Use the variable `operand` to store the number of the memory location on which the current instruction operates. Thus, if an instruction has an operand, it's the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then “pick off” the left two digits and place them in the variable `operationCode`, and “pick off” the right two digits and place them in `operand`.

When Simpletron begins execution, the special registers are initialized as follows:

<code>accumulator</code>	<code>+0000</code>
<code>instructionCounter</code>	<code>00</code>
<code>instructionRegister</code>	<code>+0000</code>
<code>operationCode</code>	<code>00</code>
<code>operand</code>	<code>00</code>

Now let's “walk through” the execution of the first SML instruction, +1009 in memory location 00. This process is called an **instruction execution cycle**.

The `instructionCounter` tells us the location of the next instruction to be performed. We *fetch* the contents of that location from `memory` by using the C statement

```
instructionRegister = memory[instructionCounter];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, and so on). A `switch` statement differentiates among the twelve operations of SML. The `switch` simulates the behavior of various SML instructions as follows (we leave the others to the reader):

- read:
`cin >> memory[operand];`
- load:
`accumulator = memory[operand];`
- add:
`accumulator += memory[operand];`
- Various branch instructions: We'll discuss these shortly.
- halt—This instruction prints “*** Simpletron execution terminated ***” then prints the name and contents of each register and the complete contents of all 100 memory locations. Such a printout is often called a **computer dump**. To help you program your dump function, the output below shows a sample dump:

REGISTERS:

accumulator	+0000
instructionCounter	00
instructionRegister	+0000
operationCode	00
operand	00

MEMORY:

	0	1	2	3	4	5	6	7	8	9
0	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
10	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
20	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
30	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
40	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
50	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
60	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
70	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
80	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
90	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000

A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. You can print leading 0s in front of an integer that is shorter than its field width by placing the 0 formatting flag before the field width in the format specifier as in "%02d". You can place a + or - sign before a value with the + formatting flag. So to produce a number of the form +0000, you can use the format specifier "%+05d".

Let's proceed with the execution of our program's first instruction, namely the +1009 in location 00. As we've indicated, the `switch` statement simulates this by performing the statement

```
cin >> memory[operand];
```

A question mark (?) should be displayed on the screen to prompt the user for input. The Simpletron waits for the user to type a value and then press the *Return* (or *Enter*) key. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Because the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the **instruction execution cycle**) begins anew with the **fetch** of the next instruction to be executed.

Now let's consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the `switch` as

```
instructionCounter = operand;
```

The conditional “branch if accumulator is zero” instruction is simulated as

```
if (accumulator == 0) {
    instructionCounter = operand;
}
```

At this point, you should implement your Simpletron simulator and run the SML programs you wrote in Exercise 7.9. You may embellish SML with additional features and provide for these in your simulator. Exercise 7.11 lists several possible embellishments.

Your simulator should check for various types of errors. During the program **loading phase**, for example, each number the user types into the Simpletron’s **memory** must be in the range -9999 to +9999. Your simulator should use a **while** loop to test that each number entered is in this range, and, if not, keep prompting the user to re-enter the number until a correct number is entered.

During the execution phase, your simulator should check for serious errors, such as attempts to **divide by zero**, attempts to execute an **invalid operation code** and **accumulator overflows** (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999). Such serious errors are fatal errors. When a fatal error is detected, print an error message such as:

```
*** Attempt to divide by zero
*** Simpletron execution abnormally terminated ***
```

and print a full computer dump in the format we’ve discussed previously. This will help the user locate the error in the program.

Implementation Note: When you implement the Simpletron Simulator, define the **memory** array and all the registers as variables in **main**. The program should contain three other functions—**load**, **execute** and **dump**. Function **load** reads the SML instructions from the user at the keyboard. (Once you study file processing in Chapter 8, you’ll be able to read the SML instruction from a file.) Function **execute** executes the SML program currently loaded in the **memory** array. Function **dump** displays the contents of **memory** and all of the registers stored in **main**’s variables. Pass the **memory** array and registers to the other functions as necessary to complete their tasks. Functions **load** and **execute** need to modify variables that are defined in **main**, so you’ll need to pass those variables to the functions by reference using pointers. You’ll need to modify the statements we showed throughout this problem description to use the appropriate pointer notations.

7.11 (Challenge Project: Modifications to the Simpletron Simulator) In this exercise, we propose several modifications and enhancements to Exercise 7.10’s Simpletron Simulator. In Exercises 13.31 and 13.32, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:

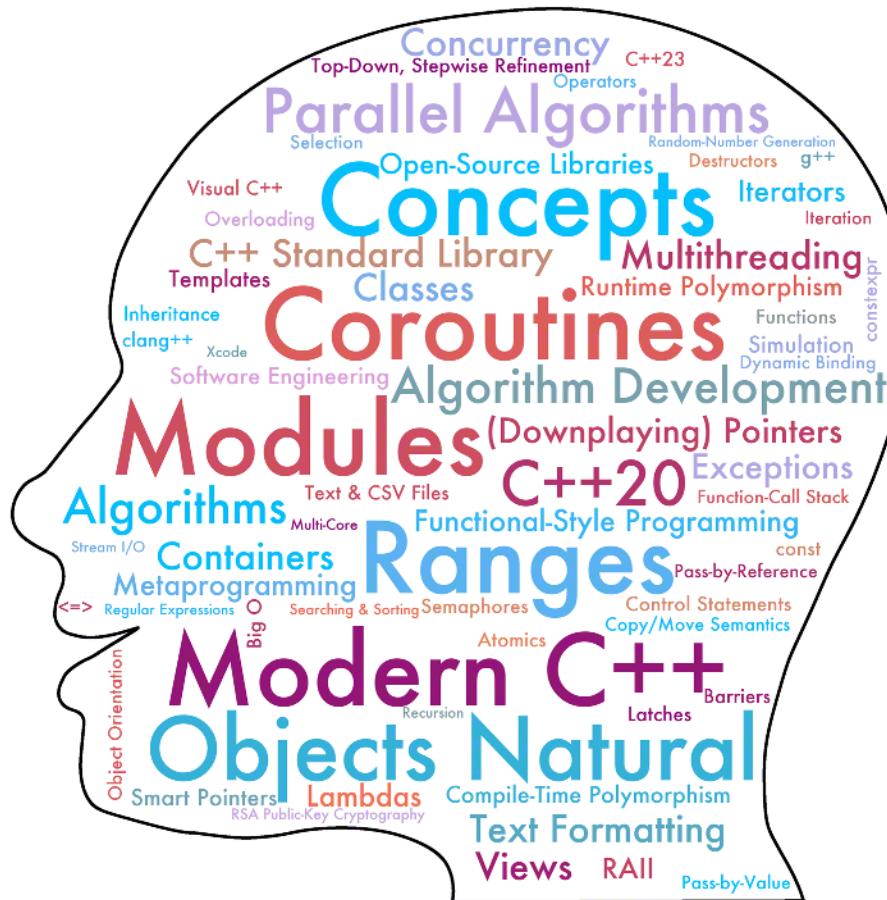
- a) Extend the Simpletron Simulator’s memory to contain 1000 memory locations (000 to 999) to enable the Simpletron to handle larger programs.
- b) Allow the simulator to perform remainder calculations. This requires an additional Simpletron Machine Language instruction.
- c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.

- d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions. See the Number Systems appendix at <https://deitel.com/cpphtp11> to learn about hexadecimal.
- e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.
- f) Modify the simulator to process floating-point values in addition to integers.
- g) Modify the simulator to detect division by 0 logic errors.
- h) Modify the simulator to detect arithmetic-overflow errors.
- i) Modify the simulator to handle string input. [*Hint:* Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store it beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to either a left or right half word.]
- j) Modify the simulator to handle output of strings stored in the format of part (g). [*Hint:* Add a machine-language instruction that prints a string beginning at a specified Simpletron memory location. The first half of the word at that location is the length of the string in characters. Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

This page intentionally left blank

8

strings, string_views, Text Files, CSV Files and Regex



Objectives

In this chapter, you'll:

- Perform various string manipulations, including assignment, concatenation, comparison, obtaining substrings, searching within strings, modifying strings and converting strings to numbers.
- Use `string_views` for lightweight views of contiguous characters.
- Write and read sequential text files.
- Input and output quoted text.
- Perform input from and output to `strings` in memory.
- Use raw string literals to automatically escape special characters in strings and to create multiline strings.
- Do an Objects-Natural data science case study using the `rapidcsv` open-source library to read and process data about the *Titanic* disaster from a CSV (comma-separated values) file.
- Do an Objects-Natural data science case study using regular expressions and C++'s `<regex>` library to search strings for patterns, validate data and replace substrings.

Outline

8.1	Introduction	8.15	Reading and Writing Quoted Text
8.2	<code>string</code> Assignment and Concatenation	8.16	Updating Sequential Files
8.3	Comparing <code>strings</code>	8.17	String Stream Processing
8.4	Substrings	8.18	Raw String Literals
8.5	Swapping <code>strings</code>	8.19	Objects-Natural Data Science Case Study: Reading and Analyzing a CSV File Containing <i>Titanic</i> Disaster Data
8.6	<code>string</code> Characteristics	8.19.1	Using <code>rapidcsv</code> to Read the Contents of a CSV File
8.7	Finding Substrings and Characters in a <code>string</code>	8.19.2	Reading and Analyzing the <i>Titanic</i> Disaster Dataset
8.8	Replacing and Erasing Characters in a <code>string</code>	8.20	Objects-Natural Data Science Case Study: Intro to Regular Expressions
8.9	Inserting Characters into a <code>string</code>	8.20.1	Matching Complete Strings to Patterns
8.10	Numeric Conversions	8.20.2	Replacing Substrings
8.11	<code>string_view</code>	8.20.3	Searching for Matches
8.12	Files and Streams	8.21	Wrap-Up
8.13	Creating a Sequential File		Exercises
8.14	Reading Data from a Sequential File		

8.1 Introduction

This chapter discusses additional `std::string` features and introduces `string_views`, text-file processing, CSV-file processing and regular expressions.

`std::string`

We've been using `std::string` objects since Chapter 2. Here, we introduce many more `std::string` manipulations, including assignments, comparisons, extracting substrings, searching for substrings and modifying `std::string` objects. We also discuss converting `std::string` objects to numeric values and vice versa.

`std::string_view`

We introduce `string_views`—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 similar capabilities to `std::strings`, making them appropriate for cases where you do not need modifiable strings.

Text Files and String Stream Processing

Data in memory is temporary. **Files** are used for **data persistence**—permanent data retention. Computers store files on **secondary storage devices**, such as flash drives, and frequently today, in the cloud. We explain how to build C++ programs that create, update and process text files. We also show how to output data to and read data from a `std::string` in memory using `ostringstreams` and `istringstream`s.



Objects-Natural Data Science Case Study: CSV Files and the Titanic Disaster Dataset

This chapter's first of three Objects-Natural case studies introduces 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.

A commonly used dataset for 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 during its first voyage of April 10–15, 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 Data Science Case Study: Using Regular Expressions to Search Strings for Patterns, Validate Data and Replace Substrings

This chapter's third Objects-Natural case study introduces regular expressions, which are particularly crucial in today's data-rich applications for

- cleaning and preparing data for analysis,
- data mining—such as locating URLs, email addresses and phone numbers in large bodies of text—and
- transforming data to other formats—such as converting tab-delimited data to comma-delimited data.

We'll use `regex` objects to create regular expressions, then use them with various functions in the `<regex>` header to match patterns in text. Earlier chapters 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 various `std::string` assignment and concatenation capabilities.

```
1 // fig08_01.cpp
2 // Demonstrating string assignment and concatenation.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6
7 int main() {
8     std::string s1{"cat"};
9     std::string s2; // initialized to the empty string
10    std::string s3; // initialized to the empty string
11
12    s2 = s1; // assign s1 to s2
13    s3.assign(s1); // assign s1 to s3
14    std::cout << std::format("s1: {}\ns2: {}\ns3: {}\n\n", s1, s2, s3);
15
16    s2.at(0) = 'r'; // modify s2
17    s3.at(2) = 'r'; // modify s3
18    std::cout << std::format("After changes:\ns2: {}\ns3: {}", s2, s3);
19}
```

Fig. 8.1 | Demonstrating string assignment and concatenation. (Part I of 2.)

```

20  std::cout << "\n\nAfter concatenations:\n";
21  std::string s4{s1 + "apult"}; // concatenation
22  s1.append("acomb"); // create "catacomb"
23  s3 += "pet"; // create "carpet" with overloaded +=
24  std::cout << std::format("s1: {}\ns3: {}\ns4: {}\n", s1, s3, s4);
25
26  // append locations 4 through end of s1 to
27  // create string "comb" (s5 was initially empty)
28  std::string s5; // initialized to the empty string
29  s5.append(s1, 4, s1.size() - 4);
30  std::cout << std::format("s5: {}\n", s5);
31 }

```

```

s1: cat
s2: cat
s3: cat

After changes:
s2: rat
s3: car

After concatenations:
s1: catacomb
s3: carpet
s4: catapult
s5: comb

```

Fig. 8.1 | Demonstrating `string` assignment and concatenation. (Part 2 of 2.)

String Assignment

Lines 8–10 create the `string` variables `s1`, `s2` and `s3`. Line 12 uses the assignment operator to copy the contents of `s1` into `s2`. Line 13 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.¹ For example, one overload copies a specified number of characters, as in

```
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 16–17 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"). You can also 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 `string` subscript operator, `[]`, does not check whether the index is in bounds. 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

1. For details of each, see https://en.cppreference.com/w/cpp/string/basic_string/assign.

```

for (char c : s3) {
    cout << c;
}

```

which ensures that you do not access elements outside the `string`'s bounds.

Accessing String Elements By Index

Line 21 initializes `s4` to the contents of `s1`, followed by "apult". For `std::string`, the `+` operator denotes string concatenation. Line 22 uses the member function `append` to concatenate `s1` and "acomb". Next, line 23 uses the overloaded addition assignment operator, `+=`, to concatenate `s3` and "pet". Then line 29 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`).



Checkpoint

1 (Code) Use `string` member function `assign` to copy the contents of the string `id1` into the string `id2`.

Answer: `id2.assign(id1);`

2 (Fill-in) A `std::string`'s `at` member function performs range checking and throws a(n) _____ exception if the index is not within the `string`'s bounds.

Answer: `out_of_range`.

3 (Code) Display `my_string`'s characters one at a time using a range-based `for` statement.

Answer:

```

for (char c : my_string) {
    std::cout << c;
}

```

8.3 Comparing strings

`std::string` provides member functions for comparing `strings` (Fig. 8.2). We call function `displayResult` (lines 7–17) throughout this example to display each comparison's result. The program declares four `strings` (lines 20–23) and outputs each (lines 25–26).

```

1 // fig08_02.cpp
2 // Comparing strings.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 void displayResult(const std::string& s, int result) {
8     if (result == 0) {
9         std::cout << std::format("{} == 0\n", s);
10    }
11    else if (result > 0) {
12        std::cout << std::format("{} > 0\n", s);
13    }

```

Fig. 8.2 | Comparing strings. (Part 1 of 2.)

```

14     else { // result < 0
15         std::cout << std::format("{} < 0\n", s);
16     }
17 }
18
19 int main() {
20     const std::string s1{"Testing the comparison functions."};
21     const std::string s2{"Hello"};
22     const std::string s3{"stinger"};
23     const std::string s4{s2}; // "Hello"
24
25     std::cout << std::format("s1: {}\ns2: {}\ns3: {}\ns4: {}\n\n",
26                             s1, s2, s3, s4);
27
28     // comparing s1 and s4
29     if (s1 > s4) {
30         std::cout << "s1 > s4\n";
31     }
32
33     // comparing s1 and s2
34     displayResult("s1.compare(s2)", s1.compare(s2));
35
36     // comparing s1 (elements 2-6) and s3 (elements 0-4)
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: Testing the comparison functions.
s2: Hello
s3: stinger
s4: Hello

s1 > s4
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

```

Fig. 8.2 | Comparing strings. (Part 2 of 2.)

Comparing Strings with the Relational and Equality Operators

`std::string` objects may be compared to one another or C-strings with the relational and equality operators. Comparisons are performed **lexicographically**—that is, based on the integer values of each character. For example, 'A' has the value 65 and 'a' has the value 97 (see the Character Set appendix at <https://deitel.com/cpphtp11>), so "Apple" would be considered less than "apple", even though they are the same word. 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` (84) has a higher numeric value than `H` (72).

Comparing Strings with Member Function `compare`

Line 34 compares `s1` to `s2` using `std::string` member function `compare`. This 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`', the result is a value greater than 0, as confirmed by the output.

The `compare` call in line 38 compares portions of `s1` and `s3` using a `compare` overload. The first two arguments (2 and 5) specify the starting index and length of the portion of `s1` ("`ting`") 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 `s3` to compare (also "`ting`"). The two pieces being compared are identical, so `compare` returns 0, as confirmed in the output.

Line 42 uses another `compare` overload to compare `s4` and `s2`. The first two arguments are the starting index and length, and 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 "`He1`" begins with the same first three letters as "`Hello`" but has fewer letters overall, "`He1`" is considered less than "`Hello`" and `compare` returns a value less than zero.



Checkpoint

- 1** (*True/False*) `std::string` objects may be compared to one another or to C-strings with the relational and equality operators—each returns a `bool`.

Answer: True.

- 2** (*True/False*) When you compare string `s1` to string `s2` using `std::string` member function `compare`, the function returns 1 if the strings are equal, 2 if `s1` is lexicographically greater than `s2`, or 3 if `s1` is lexicographically less than `s2`.

Answer: False. Actually, `compare` returns 0 if the strings are equal, a positive value if `s1` is lexicographically greater than `s2`, or a negative value if `s1` is lexicographically less than `s2`.

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 with contents copied from the source `string`. Line 8 uses member function `substr` to get a substring from `s` starting at index 3 and consisting of 4 characters.

```

1 // fig08_03.cpp
2 // Demonstrating string member function substr.
3 #include <iostream>
4 #include <string>

```

Fig. 8.3 | Demonstrating `string` member function `substr`. (Part 1 of 2.)

```

5
6 int main() {
7     const std::string s{"airplane"};
8     std::cout << s.substr(3, 4) << '\n'; // retrieve substring "plan"
9 }
```

plan

Fig. 8.3 | Demonstrating `string` member function `substr`. (Part 2 of 2.)



Checkpoint

- I (Code) Use `std::string`'s member function `substr` to get a new string containing the three characters from the beginning of `my_string`.

Answer: `std::string sub{my_string.substr(0, 3)};`

8.5 Swapping strings

`std::string`'s `swap` member function for swapping the contents of two `strings`. Figure 8.4 swaps the values of `s1` and `s2`, which are not required to have the same length.

```

1 // fig08_04.cpp
2 // Using the swap function to swap two strings.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8     std::string s1{"one"};
9     std::string s2{"two"};
10
11    std::cout << std::format("Before swap:\ns1: {}; s2: {}\n\n", s1, s2);
12    s1.swap(s2); // swap strings
13    std::cout << std::format("After swap:\ns1: {}; s2: {}\n", s1, s2);
14 }
```

Before swap:
s1: one; s2: two

After swap:
s1: two; s2: one

Fig. 8.4 | Using the `swap` function to swap two `strings`.



Checkpoint

- I (Code) Write a statement that swaps the contents of the `strings` `name1` and `name2`.

Answer: `name1.swap(name2);`

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` is always at least the `string`'s current size, though it can be greater. The exact capacity 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. In practice, the maximum size is so large in most implementations that you will not encounter this exception.

Figure 8.5 demonstrates `string` member functions for determining these characteristics. We broke the example into parts for discussion. Function `printStatistics` (lines 8–12) 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`).

```

1 // fig08_05.cpp
2 // Printing string characteristics.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 // display string statistics
8 void printStatistics(const std::string& s) {
9     std::cout << std::format(
10         "capacity: {}\nmax size: {}\nsize: {}\nempty: {}",
11         s.capacity(), s.max_size(), s.size(), s.empty());
12 }
13

```

Fig. 8.5 | Printing `string` characteristics.

The program declares empty `string` `string1` (line 15) and passes it to function `printStatistics` (line 18). This call to `printStatistics` indicates that `string1`'s initial size is 0—it contains no characters. The maximum sizes of strings are larger than you'll typically need. Object `string1` is an empty `string`, so function `empty` returns `true`.

```

14 int main() {
15     std::string string1; // empty string
16
17     std::cout << "Statistics before input:\n";
18     printStatistics(string1);
19

```

```
Statistics before input:
capacity: 15
max_size: 9223372036854775807
size: 0
empty: true
```

Line 21 inputs a string (we typed `tomato`). Line 24 calls `printStatistics` to output updated `string1` statistics. The size is now 6, and `string1` is no longer empty.

```
20     std::cout << "\n\nEnter a string: ";
21     std::cin >> string1; // delimited by whitespace
22     std::cout << std::format("The string entered was: {}\n", string1);
23     std::cout << "Statistics after input:\n";
24     printStatistics(string1);
25
```

```
Enter a string: tomato
The string entered was: tomato
Statistics after input:
capacity: 15
max_size: 9223372036854775807
size: 6
empty: false
```

Line 27 inputs another string (we typed `soup`) and stores it in `string1`, replacing "`tomato`". Line 30 calls `printStatistics` to output updated `string1` statistics. Note that the length is now 4.

```
26     std::cout << "\n\nEnter a string: ";
27     std::cin >> string1; // delimited by whitespace
28     std::cout << std::format("The string entered was: {}\n", string1);
29     std::cout << "Statistics after input:\n";
30     printStatistics(string1);
31
```

```
Enter a string: soup
The string entered was: soup
Statistics after input:
capacity: 15
max_size: 9223372036854775807
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. **How the capacity grows is implementation-defined.**

```
32 // append 46 characters to string1
33 string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
34 std::cout << std::format("\n\nstring1 is now: {}\\n", string1);
35 std::cout << "Statistics after concatenation:\\n";
36 printStatistics(string1);
37 }
```

```
string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
Statistics after concatenation:
capacity: 63
max size: 9223372036854775807
size: 50
empty: false
```

Line 38 uses the 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.

```
38     string1.resize(string1.size() + 10); // add 10 elements to string1
39     std::cout << "\\n\\nStatistics after resizing to add 10 characters:\\n";
40     printStatistics(string1);
41     std::cout << '\\n';
42 }
```

```
Statistics after resizing to add 10 characters:
capacity: 63
max size: 9223372036854775807
size: 60
empty: false
```



Checkpoint

- 1 *(Fill-in)* The maximum size of a `string` is the largest possible size it can have. If this value is exceeded, a(n) _____ exception is thrown. In practice, the maximum size is so large that it's likely you'll never encounter this exception.

Answer: `length_error`.

- 2 *(True/False)* A `string`'s capacity is the `string`'s current size.

Answer: False. Actually, a `string`'s capacity is always at least the `string`'s current size, though it can be greater. The exact capacity depends on the implementation.

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. We broke this example into parts for discussion. String `s` is declared and initialized in line 8.

```

1 // fig08_06.cpp
2 // Demonstrating the string find member functions.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8     const std::string s{"noon is 12pm; midnight is not"};
9     std::cout << "Original string: " << s;
10

```

```
Original string: noon is 12pm; midnight is not
```

Fig. 8.6 | Demonstrating the string find member functions.

Member Functions `find` and `rfind`

Lines 12–13 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.

```

11 // find "is" from the beginning and end of s
12 std::cout << std::format("\ns.find(\"is\"): {}\n{s.rfind(\"is\"): {}",
13     s.find("is"), s.rfind("is"));
14

```

```
s.find("is"): 5
s.rfind("is"): 23
```

Member Function `find_first_of`

Line 16 uses the member function `find_first_of` to locate the first occurrence in `s` of any character in "misop". The search is done from the beginning of `s`. The character 'o' is found at index 1.

```

15 // find 'o' from beginning
16 int location{s.find_first_of("misop")};
17 std::cout << std::format("\ns.find_first_of(\"misop\") found {} at {}",
18     s.at(location), location);
19

```

```
s.find_first_of("misop") found o at 1
```

Member Function `find_last_of`

Line 21 uses the 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 at index 27 of the string.

```
20 // find 'o' from end
21 location = s.find_last_of("misop");
22 std::cout << std::format("\ns.find_last_of(\"misop\") found {} at {}", 
23 s.at(location), location);
24
```

```
s.find_last_of("misop") found o at 27
```

Member Function `find_first_not_of`

Line 26 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' at index 8. Line 32 uses the 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 ';' at index 12. Line 38 uses the 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` is returned.

```
25 // find '1' from beginning
26 location = s.find_first_not_of("noi spm");
27 std::cout << std::format(
28     "\ns.find_first_not_of(\"noi spm\") found {} at {}", 
29     s.at(location), location);
30
31 // find ';' at location 12
32 location = s.find_first_not_of("12noi spm");
33 std::cout << std::format(
34     "\ns.find_first_not_of(\"12noi spm\") found {} at {}", 
35     s.at(location), location);
36
37 // search for characters not in "noon is 12pm; midnight is not"
38 location = s.find_first_not_of("noon is 12pm; midnight is not");
39
40 if (location == std::string::npos) {
41     std::cout << std::format("\n{}: not found\n",
42         "s.find_first_not_of(\"noon is 12pm; midnight is not\")");
43 }
44 }
```

```
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"): not found
```



Checkpoint

- 1 (*True/False*) `string` member functions `find` and `rfind`, search from the beginning and end of a string, respectively.

Answer: True.

- 2 (*Code*) Write a `string` member function call that finds the first vowel in the string `s`.

Answer: `s.find_first_of("aeiou")`.

- 3 (Code) Write a `string` member function call that finds the last vowel in the `string` `s`.
 Answer: `s.find_last_of("aeiou")`.

8.8 Replacing and Erasing Characters in a string

Figure 8.7 demonstrates `string` member functions for replacing and erasing characters. We broke this example into parts for discussion. Lines 9–13 declare and initialize `string1`. When string literals are separated only by whitespace, the compiler concatenates them into a single string literal. Breaking apart lengthy strings in this manner can make your code more readable.

```

1 // fig08_07.cpp
2 // Demonstrating string member functions erase and replace.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8     // compiler concatenates all parts into one string
9     std::string string1{"The values in any left subtree"
10                 "\nare less than the value in the"
11                 "\nparent node and the values in"
12                 "\nany right subtree are greater"
13                 "\nthan the value in the parent node"};
14
15 std::cout << std::format("Original string:\n{}\\n\\n", string1);
16

```

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 17 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`.

```

17     string1.erase(62); // remove from index 62 through end of string1
18     std::cout << std::format("string1 after erase:{}\\n\\n", string1);
19

```

string1 after erase:
 The values in any left subtree
 are less than the value in the

Lines 20–26 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`, which 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 25, we add 1 to `position` to continue searching from the next character's location.

```

20  size_t position{string1.find(" ")}; // find first space
21
22  // replace all spaces with period
23  while (position != std::string::npos) {
24      string1.replace(position, 1, ".");
25      position = string1.find(" ", position + 1);
26  }
27
28  std::cout << std::format("After first replacement:\n{}\n\n", string1);
29

```

```

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

```

Lines 30–37 use functions `find` and `replace` to find every period and replace every period and its following character with two semicolons extracted from the replacement string. 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.

```

30  position = string1.find(".");
31
32  // replace all periods with two semicolons
33  // NOTE: this will overwrite characters
34  while (position != std::string::npos) {
35      string1.replace(position, 2, "xxxxx;yyy", 5, 2);
36      position = string1.find(".", position + 2);
37  }
38
39  std::cout << std::format("After second replacement:\n{}\n", string1);
40

```

```
After second replacement:  
The;;values;;n;;ny;;eft;;ubtree  
are;;ess;;han;;he;;alue;;n;;he
```



Checkpoint

- 1 (*True/False*) `string` member 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.

Answer: True.

- 2 (*Fill-In*) Member function _____ returns `string::npos` when the search character is not found.

Answer: `find`.

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 index 10 of `s1`. Line 15 uses `insert` to insert "xx" before `s2`'s index 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.

```
1 // fig08_08.cpp
2 // Demonstrating std::string insert member functions.
3 #include <format>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8     std::string s1{"beginning end"};
9     std::string s2{"12345678"};
10
11    std::cout << std::format("Initial strings:\ns1: {}\ns2: {}\n\n",
12                           s1, s2);
13
14    s1.insert(10, "middle "); // insert "middle " at location 10
15    s2.insert(3, "xx", 0, std::string::npos); // insert "xx" at location 3
16
17    std::cout << std::format("Strings after insert:\ns1: {}\ns2: {}\n",
18                           s1, s2);
19 }
```

```
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.



Checkpoint

- I (Code) Write a statement that inserts a comma after "Khan" in the string "Khan Zara" so the updated string contains "Khan, Zara". Assume the string is stored in the variable name.
Answer: `name.insert(4, ",");`

8.10 Numeric Conversions

C++ provides functions that convert numeric values to **strings** and vice versa.

Converting Numeric Values to string Objects

The `to_string` function (from header `<string>`) returns the **string** representation of its numeric argument. The function is overloaded for 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

The `<string>` header provides eight functions to convert **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 conversion result 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` that's either 0 or a value in the range 2–36 representing the number's base—the default is base 10. If the base is 0, the function auto-detects the base.

So, the preceding statement is equivalent to

```
int convertedInt{stoi(s, nullptr, 10)};
```

Given a `size_t` variable named `index`, the statement

```
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 letter "h" (the first character that was not converted). See the Number Systems appendix at <https://deitel.com/cpphtp11> for an overview of the binary, octal (base 8) and hexadecimal (base 16) number systems.

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 the following `string` to a floating-point value:

```
string s{"123.45hello"};
```

The statement

```
double convertedDouble{stod(s)};
```

converts the beginning of `s` to the `double` value 123.45 and stores that value in the variable `convertedDouble`. Again, the second argument is `nullptr` by default.



Checkpoint

1 (*Fill-in*) Function _____ (header `<string>`) returns the `string` representation of its numeric argument. The function is overloaded for the fundamental numeric types.

Answer: `to_string`.

2 (*Code*) Assume the `string` `s` is defined as:

```
string s{"27amendments to US Constitution"};
```

Write a statement that converts the beginning of `s` to the `int` value 27 and stores that value in `int` variable `amendments`.

Answer: `int amendments{stoi(s)};`

8.11 `string_view`

A `string_view` (header `<string_view>`) is a read-only view of the contents of a C-string, a `string` object or a range of characters in a container, such as an `array` or `vector` of `chars`. A `string_view` cannot modify the contents it views.

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.

A `string_view` does not update automatically if the contents it views change in size after you initialize the `string_view`.

`string_views` enable many `string`-style operations on C-strings without the overhead of creating and initializing `string` objects, which copies the C-string contents. The C++ Core Guidelines state that you should prefer `string` objects if you need to “own character sequences”—for example, so you can modify a `string`’s contents.² If you need a read-only view of a contiguous sequence of characters, the C++ Core Guidelines recommend using a `string_view`.³



Creating a `string_view`

Figure 8.9 demonstrates several `string_view` features. We broke this example into parts for discussion. Line 6 includes the header `<string_view>`. Line 9 creates the `string` `s1`, and line 10 copies `s1` into the `string` `s2`. Line 11 uses `s1` to initialize a `string_view`.

```

1 // fig08_09.cpp
2 // Demonstrating string_view.
3 #include <format>
4 #include <iostream>
5 #include <string>
6 #include <string_view>
7
8 int main() {
9     std::string s1{"red"};
10    std::string s2{s1};
11    std::string_view v1{s1}; // v1 "sees" the contents of s1
12    std::cout << std::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
13 }
```

```

s1: red
s2: red
v1: red
```

Fig. 8.9 | Demonstrating `string_view`.

`string_views` “See” Changes to the Characters They View

A `string_view` does not own the sequence of characters it views but “sees” any changes to the original characters. Line 15 modifies the `std::string` `s1`. Then line 16 shows `s1`’s, `s2`’s and the `string_view` `v1`’s contents. Note that `s2` was not modified because it owns a copy of `s1`’s contents.

-
2. C++ Core Guidelines, “SL.str.1: Use `std::string` to Own Character Sequences.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-string>.
 3. C++ Core Guidelines, “SL.str.2: Use `std::string_view` or `gs1::span<char>` to Refer to Character Sequences.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-view>.

```

14 // string_views see changes to the characters they view
15 s1.at(0) = 'R'; // capitalize s1
16 std::cout << std::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
17

```

```

s1: Red
s2: red
v1: Red

```

string_views Can be Compared to std::strings or string_views

Like `strings`, `string_views` support the relational and equality operators. You also can intermix `std::strings` and `string_views` as in line 20's `==` comparisons.

```

18 // string_views are comparable with strings or string_views
19 std::cout << std::format("s1 == v1: {}\ns2 == v1: {}\n\n",
20     s1 == v1, s2 == v1);
21

```

```

s1 == v1: true
s2 == v1: false

```

string_views Can Remove a Prefix or Suffix

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 adjust the character count and, if necessary, move the pointer to `string_view`'s first remaining character. Lines 23–24 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.

```

22 // string_view can remove a prefix or suffix
23 v1.remove_prefix(1); // remove one character from the front
24 v1.remove_suffix(1); // remove one character from the back
25 std::cout << std::format("s1: {}\nv1: {}\n\n", s1, v1);
26

```

```

s1: Red
v1: e

```

string_views Are Iterable

Line 28 initializes a `string_view` from a C-string. Like `strings`, `string_views` are iterable, so you can use them with the range-based `for` statement, as in lines 30–32.

```

27 // string_views are iterable
28 std::string_view v2{"C-string"};
29 std::cout << "The characters in v2 are: ";
30 for (char c : v2) {
31     std::cout << c << " ";
32 }
33

```

The characters in v2 are: C - s t r i n g

string_views Enable Various String Operations on C-Strings

Many `string` member functions that do not modify a `string`'s contents also are defined for `string_views`. For example,

- line 35 calls `size` to determine how many characters the `string_view` can “see,”
- line 36 calls `find` to get the index of '-' in the `string_view` and
- lines 37–38 use the new C++20 `starts_with` function to determine whether the `string_view` starts with 'C'.

```
34 // string_views enable various string operations on C-Strings
35 std::cout << std::format("\n\nv2.size(): {}\n", v2.size());
36 std::cout << std::format("v2.find('-'): {}\n", v2.find('-'));
37 std::cout << std::format("v2.starts_with('C'): {}\n",
38     v2.starts_with('C'));
39 }
```

```
v2.size(): 8
v2.find('-'): 1
v2.starts_with('C'): true
```



Checkpoint

- 1 *(True/False)* Like a `span`, a `string_view` owns the data it views.

Answer: False. Neither owns the data it views.

- 2 *(True/False)* It is costly to remove characters from a `string_view`'s beginning or end.

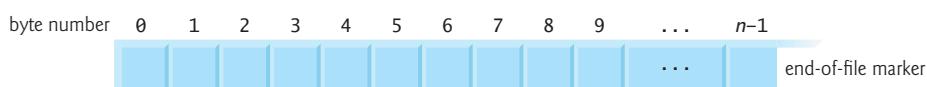
Answer: False. Actually, removing a specified number of characters from a `string_view`'s beginning or end is fast. These operations adjust the character count and, if necessary, move the pointer to the `string_view`'s first remaining character.

- 3 *(Fill-in)* Like `strings`, `string_views` are _____, so you can use them with the range-based `for` statement.

Answer: iterable.

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 `cerr` and `clog` 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,⁴ its contents are written to the standard error stream.

File-Processing Streams

File processing requires header `<fstream>`, which includes the following definitions:

- `ifstream` is for file input.
- `ofstream` is for file output.
- `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 19 also can be applied to file streams.



Checkpoint

1 (*Fill-in*) C++ views each file simply as _____.

Answer: a sequence of bytes.

2 (*Fill-in*) Each file ends either with a(n) _____ or at a specific byte number recorded in an operating-system-maintained administrative data structure.

Answer: end-of-file marker

3 (*Fill-in*) Messages written to `cerr` are output immediately, whereas messages written to `clog` are stored in a memory object called a _____. When this object is full (or flushed), its contents are written to the standard error stream.

Answer: buffer.

8.13 Creating a Sequential File

C++ imposes no structure on files. Thus, a concept like that of a record containing related data items does not exist. You structure files to meet your application's requirements. The following example shows how to 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.

4. This also occurs when the buffer is flushed, which we discuss in Chapter 19.

```
1 // fig08_10.cpp
2 // Creating a sequential file.
3 #include <cstdlib> // exit function prototype
4 #include <format>
5 #include <fstream> // contains file stream processing types
6 #include <iostream>
7 #include <string>
8
9 int main() {
10     // ofstream opens the file
11     if (std::ofstream output{"clients.txt", std::ios::out}) {
12         std::cout << "Enter the account, name, and balance.\n"
13             << "Enter end-of-file to end input.\n? ";
14
15         int account;
16         std::string name;
17         double balance;
18
19         // read account, name and balance from cin, then place in file
20         while (std::cin >> account >> name >> balance) {
21             output << std::format("{} {} {}\n", account, name, balance);
22             std::cout << "? ";
23         }
24     }
25     else {
26         std::cerr << "File could not be opened\n";
27         std::exit(EXIT_FAILURE);
28     }
29 }
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Devi 24.98
? 200 Taylor 345.67
? 300 Huber 0.00
? 400 Karimov -42.16
? 500 Sosa 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 11) and initializing it with the `filename` and the `file-open mode`. The file-open mode `ios::out` is the default for an `ofstream` object, so the second argument in line 11 is not required. Use caution when opening an existing file for output (`ios::out`). Existing files opened with mode `ios::out` are **truncated**—all data in the file is discarded without warning. If the file does not exist, the `ofstream` object creates the file.



Line 11 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 on disk, so it's placed in the same folder as the program's executable file. In legacy C++ code, the filename was specified as a pointer-based string. Modern C++ allows you to specify the filename as a `string` object.

The `<filesystem>` header provides features for manipulating files and folders. You may also specify the file to open as a `filesystem::path` object.

The following table lists the file-open modes. These modes can be combined by separating them with the `|` operator:

Mode	Description
<code>ios::app</code>	Append all output to the end of the file without modifying any data already in the file.
<code>ios::ate</code>	Open a file for output and move to the end of the file. 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 is the default action for <code>ios::out</code> .
<code>ios::binary</code>	Open a file for binary (i.e., non-text) input or output.

Testing Whether a File Was Opened Successfully

After creating an `ofstream` object and attempting to open the file, the `if` statement uses the file object `output` as a condition (line 11) to determine whether the open operation succeeded. For a file object, the overloaded operator `bool` implicitly evaluates the file object to `true` if the file opened successfully or `false` otherwise. 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
- opening a file for writing when no secondary storage space is available.



If there was an unsuccessful attempt to open the file, line 26 outputs an error message, and line 27 invokes function `exit` (from `<cstdlib>`) to terminate the program—the argument is returned to the environment that invoked the program. Passing `EXIT_SUCCESS` (from `<cstdlib>`) to `exit` indicates that the program terminated normally; passing any other value (e.g., `EXIT_FAILURE`) indicates that the program terminated due to an error.⁵

Processing Data

If line 11 opens the file successfully, the program begins processing data. Lines 12–13 prompt the user to enter the various fields for each record or the end-of-file indicator if data entry is complete. To enter the end-of-file indicator 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 20) implicitly invokes the operator `bool` 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

5. In `main`, you can return `EXIT_SUCCESS` or `EXIT_FAILURE`, rather than calling `std::exit`. When `main` terminates, its local variables are destroyed, then `main` calls `std::exit`, passing the value in `main`'s `return` statement (<https://timsong-cpp.github.io/cppwp/n4861/basic.start.main#5>). `main` implicitly returns 0 to indicate successful execution if you do not specify a `return` statement.

return `false`. You also can call the member function `eof` on the `input` object to determine whether the end-of-file indicator has been entered.

Line 20 extracts each set of data into the variables `account`, `name` and `balance`, and determines whether the end-of-file indicator 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 21 writes a set of data to the file `clients.txt`, using the stream insertion operator `<<` and the `output` object we 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 that 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, we reach the `if` statement's closing brace, so 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.

Sample Execution

In the sample execution of Fig. 8.10, we entered information for five accounts, then signaled that data entry was complete by entering the end-of-file indicator (^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.



Checkpoint

1 (True/False) C++ imposes rigid structure on files and records. This makes it convenient for you to develop file-based and record-based applications quickly.

Answer: False. Actually, C++ imposes no structure on files. Thus, a concept like that of a file of records, with each record containing related data items, does not exist. You structure files to meet your application's requirements.

2 (True/False) When you create an `ofstream` object, you must specify the `ios::out` file-open mode.

Answer: False: The `ios::out` file-open mode is the `ofstream` constructor's default second argument, so this open mode is not required.

3 (Code) Write a statement that opens the file `inventory_report.txt` without specifying the file-open mode.

Answer: `std::ofstream output{"inventory_report.txt"};`

8.14 Reading Data from a Sequential File

Figure 8.10 created a sequential file. Figure 8.11 reads data sequentially from the file `clients.txt` and displays the records. Creating an `ifstream` object opens a file for input. An `ifstream` is initialized with a filename and file-open mode. Line 11 creates an `ifstream` object called `input` that opens the `clients.txt` file for reading.⁶ If a file's con-

6. The file `clients.txt` must be in the same folder as this program's executable.

tents should not be modified, use `ios::in` to open it only for input to prevent unintentional modification of the file's contents.

```

1 // fig08_11.cpp
2 // Reading and printing a sequential file.
3 #include <cstdlib>
4 #include <format>
5 #include <fstream> // file stream
6 #include <iostream>
7 #include <string>
8
9 int main() {
10     // ifstream opens the file
11     if (std::ifstream input{"clients.txt", std::ios::in}) {
12         std::cout << std::format("{:<10}{:<13}{:>7}\n",
13             "Account", "Name", "Balance");
14
15         int account;
16         std::string name;
17         double balance;
18
19         // display each record in file
20         while (input >> account >> name >> balance) {
21             std::cout << std::format("{:<10}{:<13}{:>7.2f}\n",
22                 account, name, balance);
23         }
24     }
25     else {
26         std::cerr << "File could not be opened\n";
27         std::exit(EXIT_FAILURE);
28     }
29 }
```

Account	Name	Balance
100	Devi	24.98
200	Taylor	345.67
300	Huber	0.00
400	Karimov	-42.16
500	Sosa	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 you can omit `ios::in` when you create the `ifstream`, as in

```
std::ifstream input{"clients.txt"};
```

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 11 uses the `input` object as a condition to determine whether the file was opened successfully. The overloaded operator `bool` returns a `true` if the file was opened; otherwise, it returns `false`.

Reading from the File

Line 20 reads a set of data (i.e., a record) from the file. After line 20 executes the first time, `account` has the value 100, `name` has the value "Devi" and `balance` has the value 24.98. Each time line 20 executes, it reads another record into the variables `account`, `name` and `balance`. Lines 21–22 display each record. 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 at line 24 (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 `get` 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.

When seeking from the beginning of a file, the file-position pointer is an integer value that specifies a location as a 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 moving the `get` file-position pointer are

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);

// position n bytes forward from the current position 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` return the current locations of the `get` and `put` pointers,

respectively. The following statement assigns the *get* file-position pointer value to the variable `location` of type `std::istream::pos_type`:

```
auto location{fileObject.tellg()};
```



Checkpoint

1 (*Fill-in*) If a file's contents should not be modified, use the file open mode _____ to open it only for input to prevent unintentional modification of the file's contents.

Answer: `ios::in`.

2 (*Code*) Write a statement that repositions the *get* file-position pointer to the beginning of the file (location 0) attached to the input object `source`.

Answer: `source.seekg(0);`

3 (*Code*) Write a statement that positions the *get* file-position pointer *n* bytes forward from the current position in file object `myFile`.

Answer: `myFile.seekg(n, ios::cur);`

8.15 Reading and Writing Quoted Text

Many text files contain quoted text, such as "C++ How to Program". For example, in files representing HTML5 webpages, attribute values are enclosed in quotes. If you're building a web browser to display the contents of such a webpage, 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 "Silvio Flores" 24.98
```

Recall that the stream extraction operator `>>` treats whitespace as a delimiter. So, if we read the preceding data using the expression in line 20 of Fig. 8.11:

```
input >> account >> name >> balance
```

the first stream extraction reads 100 into the `int` variable `account`, and the second reads only "Silvio 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—"Flores"—is not a `double`.

Reading Quoted Text

The stream manipulator `quoted` (header `<iomanip>`) can read quoted text from a stream. It includes any whitespace characters in the quoted text and discards the double-quote delimiters. For example, assuming the data

```
100 "Silvio Flores" 24.98
```

the expression

```
input >> account >> std::quoted(name) >> balance
```

reads 100 into `account`, reads `Silvio Flores` as one `string` into `name` and reads 24.98 into `balance`. If the quoted data contains `\"` or `\\` escape sequences, each is read and stored in the `string` as `"` or `\`, respectively.

Writing Quoted Text

Similarly, you can write quoted text to a stream. For example, if `name` contains `Silvio Flores`, the statement

```
outputStream << std::quoted(name);
```

writes to the `outputStream`

```
"Silvio Flores"
```

If the string contains a " or \, it will be displayed as \" or \\.



Checkpoint

- 1** (*Fill-in*) The stream manipulator _____ (header `<iomanip>`) is for reading quoted text from a stream. It includes any whitespace characters in the quoted text and discards the double-quote delimiters.

Answer: `quoted`.

- 2** (*Code*) Write a statement that outputs the string "Nella Chirwa" in quotes.

Answer: `std::cout << std::quoted("Nella Chirwa");`

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 "Devi" needs to be changed to "Manirakiza," the old name cannot be overwritten without corrupting the file. The record for Devi was written to the file as

```
300 Devi 0.00
```

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

```
300 Manirakiza 0.00
```

The new record contains six more characters than the original. Any characters after "i" in "Manirakiza" would overwrite the space separating the name from 0.00, the value 0.0 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. 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 Devi 0.00 to a new file,
- write the updated record to the new file, then
- write the records after 300 Devi 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.



Checkpoint

- I (True/False) Data that is formatted and written to a sequential file cannot be modified in place without the risk of destroying other data in the file.

Answer: True.

8.17 String Stream Processing

In addition to standard stream I/O and file stream I/O, C++ 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***, both from the header **<sstream>**.

Class templates ***istringstream*** and ***ostringstream*** provide the same functionality as classes ***istream*** and ***ostream***, plus other member functions specific to in-memory formatting. 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.

To assist with data validation, C++ provides 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.

```

1 // fig08_12.cpp
2 // Using an ostringstream object.
3 #include <iostream>
4 #include <sstream> // header for string stream processing
5 #include <string>
6
7 int main() {
8     std::ostringstream output; // create ostringstream object
9
10    const std::string string1{"Output of several data types "};
11    const std::string string2{"to an ostringstream object:"};
12    const std::string string3{"\ndouble: "};
13    const std::string string4{"\n    int: "};
14
15    constexpr double d{123.4567};
16    constexpr int i{22};

```

Fig. 8.12 | Using an ***ostringstream*** object. (Part 1 of 2.)

```

17 // output strings, double and int to ostringstream
18 output << string1 << string2 << string3 << d << string4 << i;
19
20
21 // call str to obtain string contents of the ostringstream
22 std::cout << "output contains:\n" << output.str();
23
24 // add additional characters and call str to output string
25 output << "\nmore characters added";
26 std::cout << "\n\noutput now contains:\n" << output.str() << '\n';
27 }

```

```

output contains:
Output of several data types to an ostringstream object:
double: 123.457
int: 22

output now contains:
Output of several data types to an ostringstream object:
double: 123.457
int: 22
more characters added

```

Fig. 8.12 | Using an `ostringstream` object. (Part 2 of 2.)

Line 19 outputs `string string1`, `string string2`, `string string3`, `double d`, `string string4` and `int i`—all to `output` in memory. Line 22 displays `output.str()`, which returns the `string` created by `output` in line 19. Line 25 appends more data to the `string` in memory by simply issuing another stream insertion operation to `output`, then line 26 displays 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. Line 9 creates `string inputString`, which consists of characters representing two strings ("Atonia" and "test"), an `int` (123), a `double` (4.7) and a `char` ('A'). Line 10 creates the `istringstream` object `input` and initializes it to read from `inputString`. The data items in `inputString` are read into variables `s1`, `s2`, `i`, `d` and `c` in line 17, then displayed in lines 19–20. Next, we attempt to read from `input` again in line 23, but the operation fails because there is no more data in `inputString` to read. So the `input` object evaluates to `false`, and the `else` part of the `if...else` statement executes.

```

1 // fig08_13.cpp
2 // Demonstrating input from an istringstream object.
3 #include <iostream>

```

Fig. 8.13 | Demonstrating input from an `istringstream` object. (Part 1 of 2.)

```

4 #include <iostream>
5 #include <sstream>
6 #include <string>
7
8 int main() {
9     const std::string inputString{"Atonia test 123 4.7 A"};
10    std::istringstream input{inputString};
11    std::string s1;
12    std::string s2;
13    int i;
14    double d;
15    char c;
16
17    input >> s1 >> s2 >> i >> d >> c;
18
19    std::cout << "Items extracted from the istringstream object:\n"
20    << std::format("{}\n{}{}\n{}{}\n{}{}\n", s1, s2, i, d, c);
21
22    // attempt to read from empty stream
23    if (long value; input >> value) {
24        std::cout << std::format("\nlong value is: {}\n", value);
25    }
26    else {
27        std::cout << std::format("\ninput is empty\n");
28    }
29}

```

```

Items extracted from the istringstream object:
Atonia
test
123
4.7
A

input is empty

```

Fig. 8.13 | Demonstrating input from an `istringstream` object. (Part 2 of 2.)



Checkpoint

1 (*Fill-in*) C++ includes capabilities for inputting from and outputting to `strings` in memory. These capabilities often are referred to as in-memory I/O or _____.

Answer: string stream processing.

2 (*Fill-in*) _____ and _____ provide the same functionality as `istream` and `ostream`, plus other member functions specific to in-memory formatting.

Answer: `istringstream`, `ostringstream`.

3 (*Discussion*) Explain how string stream processing is used for data validation.

Answer: 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.

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

```
std::string windowsPath{"C:\\MyFolder\\MySubFolder\\MyFile.txt"};
```

For such cases, **raw string literals** are more convenient. They have the format:

```
R"(rawCharacters)"
```

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:

```
std::string windowsPath{R"(C:\\MyFolder\\MySubFolder\\MyFile.txt)"};
```

Raw strings can make code more readable, particularly when using the regular expressions we discuss in Section 8.20. Regular expressions often contain many backslash characters.

A raw string literal can include optional delimiters up to 16 characters long before the left parenthesis, (, and after the right parenthesis,), as in

```
R"MYDELIMITER(.*\d[0-35-9]-\d\d-\d\d)MYDELIMITER"
```

The optional delimiters must be identical if provided. The optional delimiters are required if the raw string literal might contain one or more right parentheses. Otherwise, the first right parenthesis encountered would be treated as the end of the raw string literal.

Raw string literals may also include line breaks, in which case the compiler inserts newline characters. For example, the raw string literal

```
R"(multiple lines  
of text)"
```

is treated as the string literal

```
"multiple lines\nof text"
```

Caution: Any indentation in the second and subsequent lines of the raw string literal is included in the raw string literal.



Checkpoint

- 1** (*Fill-in*) The compiler automatically inserts backslashes as necessary in a(n) _____ to properly escape special characters like double quotes ("), backslashes (\), etc.

Answer: raw string literal.

- 2** (*True/False*) Regular expressions often contain many backslash characters, so raw string literals make regular expressions difficult to read.

Answer: False. Actually, raw string literals make regular expressions more readable.

- 3** (*True/False*) 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 newline characters.

Answer: True.



8.19 Objects-Natural Data Science Case Study: Reading and Analyzing a CSV File Containing Titanic Disaster Data

The **CSV** (**c**omma-**s**eparated **v**alues) 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 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 following format:

```
account,name,balance
100,Devi,24.98
200,Taylor,345.67
300,Huber,0.0
400,Karimov,-42.16
500,Sosa,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. This dataset has three columns representing an `account`, `name` and `balance`.

8.19.1 Using `rapidcsv` to Read the Contents of a CSV File

The `rapidcsv`⁷ header-only library

<https://github.com/d99kris/rapidcsv>

provides the class `rapidcsv::Document` that you can use to read and manipulate CSV files.⁸ Many other libraries have built-in CSV support. For your convenience, we provided `rapidcsv` in the `examples` 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 5).

7. Copyright ©2017, Kristofer Berggren. All rights reserved.

8. Another popular data format is JavaScript Object Notation (JSON). There are C++ libraries for reading and generating JSON, such as RapidJSON (<https://github.com/Tencent/rapidjson>) and cereal (<https://uscilab.github.io/cereal/index.html>; discussed in Chapter 9's exercises).

```
1 // fig08_14.cpp
2 // Reading from a CSV file.
3 #include <iostream>
4 #include <format>
5 #include <rapidcsv.h>
6 #include <vector>
7
8 int main() {
9     rapidcsv::Document document{"accounts.csv"}; // Loads accounts.csv
10    std::vector<int> accounts{document.GetColumn<int>("account")};
11    std::vector<std::string> names{
12        document.GetColumn<std::string>("name")};
13    std::vector<double> balances{document.GetColumn<double>("balance")};
14
15    std::cout << std::format(
16        "{:<10}{:<13}{:>7}\n", "Account", "Name", "Balance");
17
18    for (size_t i{0}; i < accounts.size(); ++i) {
19        std::cout << std::format("{:<10}{:<13}{:>7.2f}\n",
20            accounts.at(i), names.at(i), balances.at(i));
21    }
22 }
```

Account	Name	Balance
100	Devi	24.98
200	Taylor	345.67
300	Huber	0.00
400	Karimov	-42.16
500	Sosa	224.62

Fig. 8.14 | Reading from a CSV file.

Line 9 creates and initializes a `rapidcsv::Document` object named `document`. This statement loads the specified file ("accounts.csv").⁹ 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 10–13 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 10's call

```
document.GetColumn<int>("account")
```

returns a `vector<int>` containing the account numbers for every record. Similarly, the calls in lines 11–12 and 13 return a `vector<string>` and a `vector<double>` containing all the records' names and balances, respectively. Lines 15–21 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 "Flores, Silvio". If this name were accidentally stored as the two strings "Flores" and

9. The file `accounts.csv` must be in the same folder as this program's executable.

"Silvio", 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

One of the most commonly used datasets for data analytics beginners is the *Titanic disaster dataset*.¹⁰ It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank during its maiden voyage of April 10–15, 1912. We'll load the dataset in Fig. 8.15, view some of its data and perform some basic data analytics.

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`.

Getting to Know the Data

In data analytics, you'll often begin by 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 1,309 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 ages are integers, but some children under 1 year of age have floating-point 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

```
https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic3info.txt
```

10. "Titanic" dataset on OpenML.org (<https://www.openml.org/d/40945>). Author: Frank E. Harrell, Jr., and Thomas Cason. Source: Vanderbilt Biostatistics (<https://biostat.app.vumc.org/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 (<https://creativecommons.org/licenses/by/4.0/>), requesting appropriate credit if you do."

Missing Data

Bad data values and missing values can significantly impact data analysis. The *Titanic* dataset is missing the ages of 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.¹¹

Loading the Dataset

Figure 8.15 uses some C++20 ranges library features we introduced in Section 6.14. We broke the program into parts for discussion purposes. Lines 15–17 create and initialize a `rapidcsv::Document` object named `titanic` that loads "titanic.csv".¹² The second and third arguments in line 16 are two of the default arguments used to initialize a `Document` object when you create it only by specifying the CSV filename. Recall from our discussion of default function arguments that when an argument is specified explicitly for a given parameter, all prior arguments in the argument list also must be specified explicitly. We provided the second and third arguments only so we could specify the fourth argument.

```
1 // fig08_15.cpp
2 // Reading the Titanic dataset from a CSV file, then analyzing it.
3 #include <format>
4 #include <algorithm>
5 #include <cmath>
6 #include <iostream>
7 #include <numeric>
8 #include <ranges>
9 #include <rapidcsv.h>
10 #include <string>
11 #include <vector>
12
13 int main() {
14     // Load Titanic dataset; treat missing age values as NaN
15     rapidcsv::Document titanic{"titanic.csv",
16         rapidcsv::LabelParams{}, rapidcsv::SeparatorParams{},
17         rapidcsv::ConverterParams{true}};
```

Fig. 8.15 | Reading the *Titanic* dataset from a CSV file, then analyzing it.

11. 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."

12. The file `titanic.csv` must be in the same folder as this program's executable.

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 ?.

Loading the Data to Analyze

Lines 20–23 use the `rapidcsv::Document`'s `GetColumn` member function to get each column by name.

```
19 // GetColumn returns column's data as a vector of the appropriate type
20 auto survived{titanic.GetColumn<int>("survived")};
21 auto sex{titanic.GetColumn<std::string>("sex")};
22 auto age{titanic.GetColumn<double>("age")};
23 auto pclass{titanic.GetColumn<int>("pclass")};
24
```

Viewing Some Rows in the Titanic Dataset

The 1,309 rows each represent one passenger. According to Wikipedia, there were approximately 1,317 passengers, and 815 of them died.¹³ For large datasets, it's possible to display only small portions of the data. A common practice when getting to know your data is to display a few rows from the beginning and end of the dataset to get a sense of the data. The code in lines 26–31 displays the first five elements of each column's data:

```
25 // display first 5 rows
26 std::cout << std::format("First five rows:\n{:<10}{:<8}{:<6}{}\n",
27 "survived", "sex", "age", "class");
28 for (size_t i{0}; i < 5; ++i) {
29     std::cout << std::format("{:<10}{:<8}{:<6.1f}{}\n",
30     survived.at(i), sex.at(i), age.at(i), pclass.at(i));
31 }
32
```

```
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
```

13. “Passengers of the *Titanic*.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Passengers_of_the_RMS_Titanic.

The code in lines 34–40 displays the last five elements of each column’s data. To determine the control variable’s starting value, line 36 calls the `rapidcsv::Document`’s `GetRowCount` member function. Then line 37 initializes the control variable to five less than the row count. Note the value `nan`, indicating a missing value for one of the age column’s rows.

```
33 // display last 5 rows
34 std::cout << std::format("\nLast five rows:\n{:<10}{:<8}{:<6}{}\n",
35     "survived", "sex", "age", "class");
36 const auto count{titanic.GetRowCount()};
37 for (size_t i{count - 5}; i < count; ++i) {
38     std::cout << std::format("{:<10}{:<8}{:<6.1f}{}\n",
39         survived.at(i), sex.at(i), age.at(i), pclass.at(i));
40 }
41
```

```
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

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 ages, and the average, minimum, maximum and median age values. Before performing these calculations, we must remove `nan` values. A calculation that includes the value `nan` produces `nan` as the calculation’s result. Lines 43–44 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, lines 45–46 create a `vector<double>` named `cleanAge`. The vector initializes its elements by iterating through the filtered results in the `removeNaN` pipeline.

```
42 // use C++20 ranges to eliminate missing values from age column
43 auto removeNaN{age |
44     std::views::filter([](const auto& x) {return !std::isnan(x);})};
45 std::vector<double> cleanAge{
46     std::begin(removeNaN), std::end(removeNaN)};
47
```

Basic Descriptive Statistics for the Cleaned Age Column

Now, we can calculate the descriptive statistics. Line 49 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 50).

```

48 // descriptive statistics for cleaned ages column
49 std::sort(std::begin(cleanAge), std::end(cleanAge));
50 size_t size{cleanAge.size()};
51 double median{};
52
53 if (size % 2 == 0) { // find median value for even number of items
54     median = (cleanAge.at(size / 2 - 1) + cleanAge.at(size / 2)) / 2;
55 }
56 else { // find median value for odd number of items
57     median = cleanAge.at(size / 2);
58 }
59
60 std::cout << "\nDescriptive statistics for the age column:\n"
61 << std::format("Passengers with age data: {}", size)
62 << std::format("Average age: {:.2f}\n", std::accumulate(
63     std::begin(cleanAge), std::end(cleanAge), 0.0) / size)
64 << std::format("Minimum age: {:.2f}\n", cleanAge.front())
65 << std::format("Maximum age: {:.2f}\n", cleanAge.back())
66 << std::format("Median age: {:.2f}\n", median);
67

```

```

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 51–58 determine the median. If `cleanAge`'s `size` is even, the median is the average of the two middle elements (line 54); otherwise, it's the middle element (line 57). Lines 60–66 display the descriptive statistics. Lines 62–63 calculate the average age by using the `accumulate` algorithm to total the ages, then dividing the result by `size`. The vector is sorted, so lines 64–65 determine the minimum and maximum values by calling the vector's `front` and `back` member functions to get the vector's first and last elements, 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—that is, a value that 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 69–73 define a lambda that counts the number of passengers in a particular class. C++20's `std::ranges::count_if` algorithm counts all the elements in its first argument, for which the lambda in its second argument returns `true`. A benefit of the `std::ranges` algorithms is that you do not need to specify the beginning and end of the container—the `std::ranges` algorithms handle this for you, simplifying your code. The first argument to `count_if` in this example will be the `vector<int>` named `pclass`. In line 72's lambda

```
[classNumber](int x) {return classNumber == x;}
```

the **lambda introducer** [`classNumber`] specifies that the `countClass` lambda's parameter `classNumber` is used in this lambda's body—this is known as **capturing** the `countClass` variable. By default, capturing is done by value, so line 72's lambda captures a copy of `classNumber`'s value. We'll say more about capturing lambdas and the `std::ranges` algorithms in Chapter 14. Lines 75–77 define constants for the three passenger classes. Lines 78–80 call the `countClass` lambda for each passenger class, and lines 82–84 display the counts.

```

68     // passenger counts by class
69     auto countClass{
70         [](const auto& column, const int classNumber) {
71             return std::ranges::count_if(column,
72                 [classNumber](int x) {return classNumber == x;});
73         };
74
75     constexpr int firstClass{1};
76     constexpr int secondClass{2};
77     constexpr int thirdClass{3};
78     const auto firstCount{countClass(pclass, firstClass)};
79     const auto secondCount{countClass(pclass, secondClass)};
80     const auto thirdCount{countClass(pclass, thirdClass)};
81
82     std::cout << "\nPassenger counts by class:\n"
83         << std::format("1st: {}\n2nd: {}\n3rd: {}\n\n",
84             firstCount, secondCount, thirdCount);
85

```

```

Passenger counts by class:
1st: 323
2nd: 277
3rd: 709

```

Basic Descriptive Statistics for the Cleaned Age Column

Let's say you want to determine some statistics about people who survived. Lines 87–89 define a lambda that counts survivors using C++20's `std::ranges::count_if` algorithm. Recall that the `survived` 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 88

```
[](auto x) {return x;}
```

simply returns the column value. If that value is 1, `count_if` counts that element. To determine how many people died, line 92 simply subtracts `survivorCount` from the `survived` vector's size. Line 94 calculates the percentage of people who survived.

```

86     // percentage of people who survived
87     const auto survivorCount{
88         std::ranges::count_if(survived, [](auto x) {return x;})
89     };
90

```

```

91     std::cout << std::format("Survived count: {}\nDied count: {}",
92                             survivorCount, survived.size() - survivorCount);
93     std::cout << std::format("Percent who survived: {:.2f}%\n\n",
94                             100.0 * survivorCount / survived.size());
95

```

```

Survived count: 500
Died count: 809
Percent who survived: 38.20%

```

Counting By Sex and By Passenger Class the Numbers of People Who Survived

Lines 97–117 iterate through the survived column, using its 1 or 0 value as a condition (line 104). For each survivor, we increment counters for the survivor’s sex (survivingFemales and survivingMales in line 105) and pclass (surviving1st, surviving2nd and surviving3rd in lines 107–115). We’ll use these counts to calculate percentages.

```

96 // count who survived by male/female, 1st/2nd/3rd class
97 int survivingMales{0};
98 int survivingFemales{0};
99 int surviving1st{0};
100 int surviving2nd{0};
101 int surviving3rd{0};
102
103 for (size_t i{0}; i < survived.size(); ++i) {
104     if (survived.at(i)) {
105         sex.at(i) == "female" ? ++survivingFemales : ++survivingMales;
106
107         if (firstClass == pclass.at(i)) {
108             ++surviving1st;
109         }
110         else if (secondClass == pclass.at(i)) {
111             ++surviving2nd;
112         }
113         else { // third class
114             ++surviving3rd;
115         }
116     }
117 }
118

```

Calculating Percentages of People Who Survived

Lines 120–129 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.

```
119 // percentages who survived by male/female, 1st/2nd/3rd class
120 std::cout << std::format("Female survivor percentage: {:.2f}%\n",
121     100.0 * survivingFemales / survivorCount)
122     << std::format("Male survivor percentage: {:.2f}%\n\n",
123         100.0 * survivingMales / survivorCount)
124     << std::format("1st class survivor percentage: {:.2f}%\n",
125         100.0 * surviving1st / survivorCount)
126     << std::format("2nd class survivor percentage: {:.2f}%\n",
127         100.0 * surviving2nd / survivorCount)
128     << std::format("3rd class survivor percentage: {:.2f}%\n",
129         100.0 * surviving3rd / survivorCount);
130 }
```

```
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%
```



Checkpoint

1 (*Fill-in*) The first row of a CSV file typically contains _____.

Answer: column names.

2 (*Fill-in*) In a CSV file representing a dataset, the rows contain the data and the columns are often called the dataset's _____.

Answer: features.

3 (*True/False*) A benefit of the `std::ranges` algorithms is that you do not need to specify the beginning and end of the container—the `std::ranges` algorithms handle this for you, simplifying your code.

Answer: True.

8.20 Objects-Natural Data Science Case Study: Intro to Regular Expressions

Sometimes you'll need to recognize patterns in text, like phone numbers, e-mail addresses, ZIP codes, webpage 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 used in each group of digits.

You'll rarely need to create regular expressions for common items like these. The following free websites

- <https://regex101.com>
- <https://regexpr.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*)—for example, locating all URLs in a webpage.
- Clean data—for example, 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—for example, 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.



Checkpoint

1 (*True/False*) A regular expression string describes a search pattern for matching characters in other strings.

Answer: True.

2 (*Fill-in*) Extracting data from text is known as _____.

Answer: scraping.

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.¹⁴ Figure 8.16 demonstrates matching entire strings to regular expression patterns. We broke the program into parts for discussion purposes.

14. Some C++ programmers prefer using third-party regular-expression libraries, such as RE2 or PCRE. For a list of other C++ regular expression libraries, see <https://github.com/fffaraz/awesome-cpp#regular-expression>.

Matching Literal Characters

The `<regex>` function `regex_match` 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 9 creates a `regex` object `r1` for the pattern "02215", containing only literal digits that match themselves in the specified order. Line 12 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**.

```

1 // fig08_16.cpp
2 // Matching entire strings to regular expressions.
3 #include <iostream>
4 #include <iomanip>
5 #include <regex>
6
7 int main() {
8     // fully match a pattern of literal characters
9     std::regex r1{"02215"};
10    std::cout << "Matching against: 02215\n"
11    << std::format("02215: {}; 51220: {}\\n\\n",
12                  std::regex_match("02215", r1), std::regex_match("51220", r1));
13

```

```

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 15), `\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 18's second call to `regex_match` returns `false` because "9876" contains only four consecutive digits.

```
14 // fully match five digits
15 std::regex r2{R"(\d{5})"};
16 std::cout << R"(Matching against: \d{5})" << "\n"
17     << std::format("02215: {}; 9876: {}\\n\\n",
18         std::regex_match("02215", r2), std::regex_match("9876", r2));
19
```

```
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. Line 21 defines a custom character class to validate a simple first name with no spaces or punctuation.

```
20 // match a word that starts with a capital letter
21 std::regex r3{"[A-Z][a-z]*"};
22 std::cout << "Matching against: [A-Z][a-z]*\\n"
23     << std::format("Angel: {}; tina: {}\\n\\n",
24         std::regex_match("Angel", r3), std::regex_match("tina", r3));
25
```

```
Matching against: [A-Z][a-z]*
Angel: true; tina: false
```

A first name might contain many letters. In the regular expression `r3` (line 21), `[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 "Vivaan", "An" 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 27) matches any character that's not a lowercase letter.

```

26 // match any character that's not a lowercase letter
27 std::regex r4{"[^a-z]"};
28 std::cout << "Matching against: [^a-z]\n"
29     << std::format("A: {}; a: {}\\n\\n",
30                     std::regex_match("A", r4), std::regex_match("a", r4));
31

```

```

Matching against: [^a-z]
A: true; a: false

```

Metacharacters in a custom character class are treated as the characters themselves. So `[*+$]` (line 33) matches a single *, + or \$ character.

```

32 // match metacharacters as literals in a custom character class
33 std::regex r5{"[*+$]"};
34 std::cout << "Matching against: [*+$]\\n"
35     << std::format("*: {}; !: {}\\n\\n",
36                     std::regex_match("*", r5), std::regex_match("!", r5));
37

```

```

Matching against: [*+$]
*: true; !: false

```

* vs. + Quantifier

To require at least one lowercase letter in a first name, replace the * quantifier in line 21 with + (line 39). This matches at least one occurrence of a subexpression to its left.

```

38 // matching a capital letter followed by at least one lowercase letter
39 std::regex r6{"[A-Z][a-z]+"};
40 std::cout << "Matching against: [A-Z][a-z]+\n"
41     << std::format("Angel: {}; T: {}\\n\\n",
42                     std::regex_match("Angel", r6), std::regex_match("T", r6));
43

```

```

Matching against: [A-Z][a-z]+
Angel: true; E: false

```

Both * and + are **greedy**—they match as many characters as possible. So the regular expression `[A-Z][a-z]+` matches any word that begins with a capital letter followed by at least one lowercase letter. You can make * and + **lazy**, so they match as few characters as possible by appending a question mark (?) to the quantifier, as in `*?` or `+?`.

Other Quantifiers

The ? quantifier by itself matches zero or one occurrence of the subexpression to its left. In the regular expression `labe11?ed` (line 45), the subexpression is the literal character "1". So in the `regex_match` calls in lines 48–49, the regular expression matches `labe11ed` (the U.K. English spelling) and `labe1ed` (the U.S. English spelling), but in the `regex_match` call in line 50, the regular expression does not match the misspelled word `labe11ed`.

```

44 // matching zero or one occurrences of a subexpression
45 std::regex r7{"label1?ed"};
46 std::cout << "Matching against: label1?ed\n"
47     << std::format("labelled: {}; labeled: {}; labelled: {}\\n\\n",
48                     std::regex_match("labelled", r7),
49                     std::regex_match("labeled", r7),
50                     std::regex_match("Label11ed", r7));
51

```

```

Matching against: label1?ed
labelled: true; labeled: true; labelled: false

```

You can use the **{*n*,*1*}** quantifier to match at least *n* occurrences of a subexpression to its left. The regular expression in line 53 matches strings containing at least three digits.

```

52 // matching n (3) or more occurrences of a subexpression
53 std::regex r8{R"(\d{3,})"};
54 std::cout << R"(Matching against: \d{3,})" << "\\n"
55     << std::format("123: {}; 1234567890: {}; 12: {}\\n\\n",
56                     std::regex_match("123", r8),
57                     std::regex_match("1234567890", r8),
58                     std::regex_match("12", r8));
59

```

```

Matching against: \d{3,}
123: true; 1234567890: true; 12: false

```

You can use the **{*n*,*m*}** quantifier to match between *n* and *m* (inclusive) occurrences of a subexpression. The regular expression in line 61 matches strings containing 3 to 6 digits.

```

60 // matching n to m inclusive (3-6), occurrences of a subexpression
61 std::regex r9{R"(\d{3,6})"};
62 std::cout << R"(Matching against: \d{3,6})" << "\\n"
63     << std::format("123: {}; 123456: {}; 1234567: {}; 12: {}\\n",
64                     std::regex_match("123", r9), std::regex_match("123456", r9),
65                     std::regex_match("1234567", r9), std::regex_match("12", r9));
66 }

```

```

Matching against: \d{3,6}
123: true; 123456: true; 1234567: false; 12: false

```



Checkpoint

- 1 (Code) Write a custom character class that matches only lowercase consonants.
Answer: [b-df-hj-np-tv-z].

- 2 (Code) Write a quantifier that would match at least 7 occurrences of the subexpression to its left.
Answer: {7,}.

- 3 (Code)** Write a quantifier that would match a minimum of 3 and a maximum of 5 occurrences of the subexpression to its left.

Answer: $\{3,5\}$.

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 13) 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 (",").

It returns a new `string` containing the modifications. The expression

```
std::regex{"\t"}
```

creates a temporary `regex` object, initializes it and immediately passes it into the function `regex_replace`. This is useful if you do not need to reuse a `regex` multiple times.

```

1 // fig08_18.cpp
2 // Regular expression replacements.
3 #include <iostream>
4 #include <iomanip>
5 #include <regex>
6 #include <string>
7
8 int main() {
9     // replace tabs with commas
10    std::string s1{"1\t2\t3\t4"};
11    std::cout << std::format("Original string: {}\n", R"(1\t2\t3\t4)")
12        << std::format("After replacing tabs with commas: {}\n",
13                      std::regex_replace(s1, std::regex{"\t"}, ","));
14 }
```

```
Original string: 1\t2\t3\t4
After replacing tabs with commas: 1,2,3,4
```

Fig. 8.17 | Regular expression replacements.

8.20.3 Searching for Matches

You can match a substring within a string using the 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`, use an `smatch` (pronounced “ess match”).
- For searches on C-strings and string literals, use a `cmatch` (pronounced “see match”).

The `<regex>` header has not yet been updated for `string_views`.

Finding a Match Anywhere in a String

The calls to function `regex_search` in lines 14–16 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 returns `true` or `false` to indicate a match.

```

1 // fig08_19.cpp
2 // Matching patterns throughout a string.
3 #include <iostream>
4 #include <iomanip>
5 #include <regex>
6 #include <string>
7
8 int main() {
9     // performing a simple match
10    std::string s1{"Programming is fun"};
11    std::cout << std::format("s1: {}\n\n", s1);
12    std::cout << "Search anywhere in s1:\n"
13        << std::format("Programming: {}; fun: {}; fn: {} \n\n",
14                      std::regex_search(s1, std::regex{"Programming"}),
15                      std::regex_search(s1, std::regex{"fun"}),
16                      std::regex_search(s1, std::regex{"fn"}));
17

```

```

s1: Programming is fun
Search anywhere in s1:
Programming: true; fun: true; fn: false

```

Fig. 8.18 | Matching patterns throughout a string.

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. Using the constant `regex_constants::icase`, you can perform a case-insensitive search.

```

18 // ignoring case
19 std::string s2{"AZUEL MAGAR"};
20 std::smatch match; // store the text that matches the pattern
21 std::cout << std::format("s2: {}\n\n", s2);
22 std::cout << "Case insensitive search for Azuel in s2:\n"
23     << std::format("Azuel: {} \n", std::regex_search(s2, match,
24                     std::regex{"Azuel", std::regex_constants::icase}))
25     << std::format("Matched text: {} \n\n", match.str());
26

```

```
s2: AZUEL MAGAR
```

```
Case insensitive search for Azuel in s2:  
Azuel: true  
Matched text: AZUEL
```

Lines 23–24 call `regex_search`'s three-argument version, in which the arguments are

- the string to search (`s2`; line 23), which in this case contains all capital letters,
- the `smatch` object (`match`; line 23), which stores the match if there is one, and
- the `regex` pattern to match (line 24).

Here, we created the `regex` with a second argument

```
std::regex{"Azuel", std::regex_constants::icase}
```

This `regex` matches the literal characters "Azuel" regardless of their case. So, in `s2`, "AZUEL" matches the `regex` "Azuel" because both have the same letters, even though "AZUEL" contains only uppercase letters. To confirm the matching text, line 25 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` (lines 28–29) that matches the `regex` `phone` (line 30) and displays the matching text.

```
27 // finding all matches
28 std::string contact{
29     "Ando Devi, Home: 555-555-1234, Work: 555-555-4321";
30     std::regex phone{R"(\d{3}-\d{3}-\d{4})"};
31
32     std::cout << std::format("Finding phone numbers in:\n{}\\n", contact);
33     while (std::regex_search(contact, match, phone)) {
34         std::cout << std::format("    {}\\n", match.str());
35         contact = match.suffix();
36     }
37 }
```

```
Finding phone numbers in:  
Ando Devi, 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`.



Checkpoint

1 (*Fill-in*) You can match a substring within a string using function _____ which returns `true` if any part of an arbitrary `string` matches the regular expression.

Answer: `regex_search`.

2 (*True/False*) The `regex_constants` can customize how regular expressions perform matches. The constant `regex_constants::icase` enables case-insensitive searches.

Answer: True.

8.21 Wrap-Up

In this chapter, we presented additional `string` features and introduced `string_views`, text-file processing, CSV-file processing, ZIP-file processing and regular expressions. We introduced many more `std::string` manipulations, including assignments, comparisons, extracting substrings, swapping, determining string characteristics, searching for substrings and modifying strings. We also discussed converting `std::string` objects to numeric values and vice versa.

We introduced `string_views` for creating read-only views of C-strings and `string` objects. You saw that, like spans, `string_views` do not own the data they view. You'll see that `string_views` have similar capabilities to `std::strings`, making them appropriate where you do not need modifiable strings.

We pointed out that data in memory is temporary and that files can be used for permanent data retention. Then, we showed how to create and process text files. We presented similar techniques for outputting data to and reading data from `std::strings` in memory using `ostringstreams` and `istringstream`s. We used raw string literals to automatically escape special characters in strings and to create multiline strings.

In the first of this chapter's three Objects-Natural case studies, we introduced the CSV (comma-separated values) file format, which is commonly used to store datasets. We used the `rapidcsv` open-source library to read the contents of the *Titanic* dataset from a CSV file, then performed some basic data analytics. In the second Objects-Natural case study, we introduced the ZIP archive file format for compressing files and folders into a single file, typically for efficient distribution over the web. In the third Objects-Natural case study, we introduced regular expressions and their common applications, such as cleaning and preparing data for analysis, data mining and transforming data to other formats. We created `regex` objects representing regular expressions, then used them with various `<regex>` header functions to match patterns in text.

We've now introduced the basic concepts of control statements, functions, arrays, vectors, `strings` and files. You've already seen in many of our examples and Objects-Natural case studies that a C++ application typically creates and manipulates objects to perform its work. In Chapter 9, you'll learn how to implement your own custom classes, then create and use objects of those classes in applications. We'll begin discussing class design and related software-engineering concepts.

String Exercises

8.1 (*Word Endings*) Write a program that reads in several `strings` and prints only those ending in "r" or "ay". Only lowercase letters should be considered.

8.2 (string Concatenation) Write a program that separately inputs a first and last name and concatenates the two with a space between them into a new `string`. Show two techniques for accomplishing this.

8.3 (Printing a string Backward) Write a program that inputs a `string` and prints it backward. Convert all uppercase characters to lowercase and all lowercase characters to uppercase.

8.4 (Recognizing Palindromes) A palindrome is a word or phrase that reads the same backward and forward. For example, "radar" and "noon" are palindromes. Write a program that inputs a line of text and displays whether the text is a palindrome. Ignore whitespace and punctuation. Test your program with the following famous phrase referring to Theodore Roosevelt and his plan to complete the Panama canal:¹⁵

a man a plan a canal panama

8.5 (Pig Latin) Write a program that encodes English-language phrases into a form of coded language called pig Latin.¹⁶ For simplicity, use the following algorithm to form a pig Latin phrase from an English-language phrase: Tokenize (that is, break apart) the phrase into words. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus, the word "jump" becomes "umpjay," the word "the" becomes "hetay," and the word "computer" becomes "omputercay." If the word starts with a vowel, just add "ay." Blanks between words remain as blanks, and single-letter words are not modified. Assume the following: Each phrase consists of words separated by blanks and there are no punctuation marks. Enable the user to enter a sentence, then display the sentence in pig Latin.

8.6 (Reversing a Sentence) Write a program that reads a line of text as a string, then outputs the words in reverse order. For example,

welcome to C++ programming

would be output as

programming C++ to welcome

8.7 (Displaying Strings That Start with b) Write a program that reads a line of text one word at a time and outputs only those words beginning with the letter b.

8.8 (Displaying Strings That End with ed) Write a program that reads a line of text one word at a time and outputs only those words ending with the letters ed.

8.9 (Converting Integers to Characters) The C++20 `format` function can use the `c` presentation type to format an integer as a character. For example, the following statement would output the value 65 as its character equivalent, capital A:

```
std::cout << std::format("{:c}\n", 65);
```

Use this technique to create a table of the character codes in the range 0 to 255 and their corresponding Unicode characters.

15. "Leigh Mercer." Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Leigh_Mercer.

16. "Pig Latin." Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Main_Page.

8.10 (Converting Integers to Emojis) Modify the previous exercise to display 10 emojis beginning with the smiley face, which has the value 0x1F600:¹⁷



The value 0x1F600 is a hexadecimal (base 16) integer. See the Number Systems appendix at <https://deitel.com/cpphtp11> for information on the hexadecimal number system. You can find emoji codes by searching online for “Unicode full emoji list.”¹⁸ The Unicode website precedes each character code with “U+” (representing Unicode). Replace “U+” with “0x” to properly format the code as a hexadecimal integer.

8.11 (Simple Encryption) A simple encryption algorithm known as “rot13” rotates each alphabetic character by 13 positions in the alphabet. Thus, 'a' corresponds to 'n', and 'x' corresponds to 'k'. rot13 is an example of **symmetric key encryption**. With symmetric key encryption, both the encrypter and decrypter use the same private key.

- a) Write a program that encrypts a message using rot13.
- b) Write a program that decrypts the scrambled message using 13 as the key.
- c) After writing the programs of part (a) and part (b), briefly answer the following question: If you did not know the key for part (b), how difficult do you think it would be to break the code? Exercise 8.12 asks you to write a program to accomplish this.

8.12 (Simple Decryption) In Exercise 8.11, you wrote a simple encryption program using rot13 symmetric key encryption. Write a program that will attempt to decrypt a “rot13” message using simple frequency substitution. (Assume that you do not know the key.) The most frequent letters in the encrypted phrase should be replaced with the most commonly used English letters (a, e, i, o, u, s, t, r, etc.). Write the possibilities to a file. What made the code-breaking easy? How can the encryption mechanism be improved?

8.13 (Cryptograms) Write a program that creates a cryptogram out of a `string`. A cryptogram is a message or word in which each letter is replaced with another letter. For example, the `string`

The bird was named squawk

might be scrambled to form

cin vrjs otz ethns zxqtop

Spaces are not scrambled. In this case, 'T' was replaced with 'x', each 'a' was replaced with 'h', etc. Uppercase letters become lowercase letters in the cryptogram. Use techniques similar to those in Exercise 8.11.

8.14 (Solving Cryptograms) Modify Exercise 8.13 to allow the user to solve the cryptogram. The user should input two characters at a time—the first specifies a letter in the cryptogram, and the second specifies the replacement letter. If the replacement letter is correct, replace the letter in the cryptogram with the replacement letter in uppercase.

17. The look-and-feel of emojis varies across systems. The emoji shown here is from macOS. Also, the emoji symbols might not display correctly, depending on your system’s fonts.

18. “Full Emoji List, v15.0” Accessed April 14, 2023. <https://unicode.org/emoji/charts/full-emoji-list.html>.

File Exercises

8.15 (*Class Average: Writing Grades to a Plain Text File*) Figure 3.2 presented a class-average program in which you could enter any number of grades followed by a sentinel value, then calculate the class average. Another approach would be to read the grades from a file. Write a program that enables you to store any number of grades into a `grades.txt` plain text file.

8.16 (*Class Average: Reading Grades from a Plain Text File*) Write a program that reads the grades from the `grades.txt` file you created in the previous exercise. Display the individual grades and their total, count and average.

8.17 (*Class Average: Reading Student Records from a CSV File*) An instructor teaches a class in which each student takes three exams. The class's information is stored in a file named `grades.csv`. Each record has the following format:

`firstname, lastname, exam1grade, exam2grade, exam3grade`

Use the techniques you learned in Section 8.19 to read the `grades.csv`. Display the data in tabular format.

8.18 (*Class Average: Creating a Grade Report from a CSV File*) Modify your solution to the preceding exercise to create a grade report that displays each student's average to the right of that student's row and the class average for each exam below that exam's column.

8.19 (*Telephone-Number Word Generator Enhancement*) Modify the program you wrote in Exercise 6.19 to write to a file every possible seven-letter word corresponding to a seven-digit phone number. You can then conveniently browse through this file with a text editor to look for recognizable words.

8.20 (*`sizeof` Operator*) Write a program that uses the `sizeof` operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file `datasize.txt` so that you may print the results later. The results should be formatted as two columns with the type name in the left column and the type's size in the right column, as shown below:

<code>char</code>	1
<code>unsigned char</code>	1
<code>short int</code>	2
<code>unsigned short int</code>	2
<code>int</code>	4
<code>unsigned int</code>	4
<code>long int</code>	4
<code>unsigned long int</code>	4
<code>long long int</code>	8
<code>unsigned long long int</code>	8
<code>float</code>	4
<code>double</code>	8
<code>long double</code>	10

We listed the integer types from smallest to largest size, followed by the floating-point types from smallest to largest size. The sizes of these types on your computer and compiler might differ from those listed here.

Regular Expression Exercises

8.21 (*Regular Expressions: Condense Spaces to a Single Space*) Check whether a sentence contains more than one space between words. If so, remove the extra spaces and display the results. For example, "Hello World" should become "Hello World".

8.22 (*Regular Expressions: Capturing Substrings*) Reimplement Exercise 8.7 using regular expressions that capture the matching substrings, then display them.

8.23 (*Regular Expressions: Counting Characters and Words*) Use regular expressions to count the number of digits, non-digit characters, whitespace characters and words in a string.

8.24 (*Regular Expressions: Matching Numeric Values*) Write a regular expression that searches a string and matches a valid number. A number can have any number of digits, but it can have only digits and a decimal point and possibly a leading sign. The decimal point is optional, but if it appears in the number, there must be only one with digits on its left and right. There should be whitespace or a beginning or end-of-line character on either side of a valid number.

8.25 (*Regular Expression: Password Format Validator*) Search online for secure password recommendations, then research existing regular expressions that validate secure passwords. Two examples of password requirements are:

- Passwords must contain at least five words, each separated by a hyphen, a space, a period, a comma or an underscore.
- Passwords must have a minimum of 8 characters and contain at least one each from uppercase characters, lowercase characters, digits and punctuation characters (such as characters in "!@#\$%^&*?").

Write regular expressions for each of the two requirements above, then use them to test sample passwords.

8.26 (*Regular Expressions: Testing Regular Expressions Online*) Before using any regular expression in your code, you should thoroughly test it to ensure it meets your needs. Use a regular expression website like regex101.com to explore and test existing regular expressions, then write your own regular expression tester.



8.27 (*Regular Expressions: Munging Dates*) In data science, preparing data for analysis is called **data munging** or **data wrangling**. Dates are stored and displayed in several common formats. Three common month-day-year formats are

042555
04/25/1955
April 25, 1955

Use regular expressions to search a string containing dates, find substrings that match these formats and munge them into the other formats. The original string should have one date in each format, so there will be six transformations.

More Challenging String- and File-Manipulation Exercises

This section contains intermediate and advanced string and file manipulation exercises. You should find these problems challenging and, hopefully, entertaining. The problems vary considerably in difficulty. Some require an hour or two of coding. Others are useful

for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects. Some are appropriate for thesis topics.

8.28 (*Project: Metric Conversions*) Write a program that assists the user with some common metric-to-United-States-customary-system (USCS or USC) conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, and so on for the metric system and inches, quarts, pounds, and so on for the USCS system) and should respond to simple questions, such as

How many inches are in 2 meters?
How many liters are in 10 quarts?

Your program should recognize invalid conversions. For example, the following question is not meaningful because "feet" is a unit of length and "kilograms" is a unit of mass:

How many feet are in 5 kilograms?

Assume that all questions are in the form shown above. Use regular expressions to capture the important substrings, such as "inches", "2" and "meters" in the first sample question above.

8.29 (*Project: Spam Scanner*) Spam (or junk e-mail) costs organizations billions of dollars annually in spam-prevention software, equipment, network resources, bandwidth, and lost productivity. Research online some common spam e-mail messages and words, and check your junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write a program in which the user enters an e-mail message. Read the message into a string. Then search the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam based on the number of points it received.

8.30 (*Project: Evaluate Word Problems*) Write a program that enables the user to enter mathematical word problems like "two times three," and "seven minus five." Use string processing to break apart the string into the numbers and the operation and return the result. So "two times three" would return 6 and "seven minus five" would return 2. To keep things simple, assume the user enters only the words for the numbers 0 through 9 and only the operations "plus", "minus", "times" and "divided by".

8.31 (*Research: Inter-Language Translation*) This exercise will help you explore one of the most challenging problems in natural language processing and artificial intelligence. The Internet brings us all together in ways that make inter-language translation particularly valuable.

With advances in machine learning, artificial intelligence and natural language processing, services like Google Translate (130+ languages¹⁹) and Bing Microsoft Translator (100+ languages²⁰) offer free-tier and paid-tier services that can translate between languages instantly.

One challenge faced by the natural language translation community is that different translation tools may yield different results. To get a sense of this, use Bing Microsoft

19. "Google Translate Learns 24 New Languages." Accessed April 14, 2023. <https://blog.google/products/translate/24-new-languages/>.

20. "Microsoft Translator: Features." Accessed April 14, 2023. <https://www.microsoft.com/en-us/translator/languages/>.

Translator and Google Translate (and others of your choice) to perform the following tasks:

- Start with an English phrase such as “Out of sight, out of mind.”
- Translate that sentence from one language to another and eventually back to English. For example, you might translate from English, to Spanish, to French, to Italian, to Chinese, to Russian, to Japanese, then back to English.

Do you get the original sentence?



8.32 (Data Science Project: Working with the diamonds.csv Dataset) Data scientists work extensively with datasets, many in CSV format. Datasets are available for almost anything you’d want to study. There are numerous dataset repositories from which you can download datasets in CSV and other formats, including

<https://vincentarelbundock.github.io/Rdatasets/datasets.html>

and

<https://github.com/awesomedata/awesome-public-datasets>

The Kaggle competition site (owned by Google):²¹

<https://www.kaggle.com/datasets?filetype=csv>

has over 50,000 public datasets. The U.S. government’s data.gov site:

https://catalog.data.gov/dataset?res_format=CSV&_res_format_limit=0

has approximately 250,000 datasets, with over 21,500 in CSV format.

In this exercise, you’ll use the diamonds dataset to perform tasks similar to those you saw in Section 8.19. This is available as `diamonds.csv` from various sources, including the Kaggle and Rdatasets sites listed above. The dataset contains information on 53,940 diamonds, including each diamond’s carats, cut, color, clarity, depth, table (flat top surface), price and *x*, *y* and *z* measurements. The Kaggle site’s web page for this dataset describes each column’s content.²²

Perform the following tasks to study and analyze the diamonds dataset:

- Download `diamonds.csv` from one of the dataset repositories.
- Use the `rapidcsv` techniques you learned in Section 8.19 to load the dataset’s columns into vectors.
- Display the dataset’s first seven rows.
- Display the dataset’s last seven rows.
- Calculate and display the descriptive statistics for the numerical columns—`carat`, `depth`, `table`, `price`, *x*, *y* and *z*. For each column, show the count, average, minimum, maximum and median.
- You can also calculate descriptive statistics for the dataset’s non-numeric columns—`cut`, `color` and `clarity`—each containing text. Such columns are known as **categorical data**. For the `cut`, `color` and `clarity` columns, display each column’s count, number of unique values, most frequent value and number of occurrences of the most frequent value.
- For each categorical data column, display its unique category values.

21. To download data from Kaggle, you must register for a free account. This is true of various other dataset repository sites as well.

22. <https://www.kaggle.com/shivam2503/diamonds>.



8.33 (Data Science Project: Working with the Iris.csv Dataset) Another popular dataset for data analytics novices is the Iris dataset, which contains 150 records of information about three Iris plant species. Like the diamonds dataset, the Iris dataset is available from various online sources, including Kaggle. Investigate the Iris dataset's columns,²³ then perform the following tasks to study and analyze the dataset:

- a) Download Iris.csv from one of the dataset repositories.
- b) Use the `rapidcsv` techniques you learned in Section 8.19 to load the dataset's columns into vectors.
- c) Display the dataset's first five rows.
- d) Display the dataset's last five rows.
- e) Calculate and display the descriptive statistics for the numerical data columns—`SepalLengthCm`, `SepalWidthCm`, `PetalLengthCm` and `PetalWidthCm`. For each column, show the count, average, minimum, maximum and median.

8.34 (Project: State-of-the-Union Speeches) Text files of all U.S. Presidents' State-of-the-Union speeches are available online.²⁴ (Note: These speeches are generally written for presidents by professional speech writers.) Download one of these speeches. Write a program that reads the speech from the file, then displays statistics about the speech, including the total word count, the total character count, the average word length, the average sentence length, a word distribution of the words frequencies, and the top 10 longest words. "Natural Language Processing (NLP)" tools provide sophisticated techniques for analyzing and comparing such texts. Additional Project: Download all the Presidents' State-of-the-Union speeches and print their comparative statistics.

8.35 (Research: Grammarly) Copy and paste State of the Union speeches into the free version of Grammarly or similar software. Compare the reading grade levels for speeches from several presidents.

8.36 (Project: Simple Sentiment Analysis) Search online for files of positive words (like happy, pleasant, ...) and files of negative words (like sad, angry, ...). Create a program that inputs text, then searches through that text to see how many positive and negative words it contains. Based on those counts, rate the text as positive, negative or neutral. Test your program with the text of your social media posts and those of your friends. Sentiment analysis is a popular application of natural language processing (NLP).

8.37 (Project: Crossword-Puzzle Generator) Most people have tried crossword puzzles, but few have attempted to generate one by hand. Generating a crossword puzzle is suggested here as a string-manipulation and file-processing project requiring substantial sophistication and effort. Consider searching for crossword-puzzle generators online and studying them before attempting this project.

23. <https://www.kaggle.com/uciml/iris/home>.

24. "Annual Messages to Congress on the State of the Union (Washington 1790 - the present)." Accessed April 14, 2023. <https://www.presidency.ucsb.edu/documents/presidential-documents-archive-guidebook/annual-messages-congress-the-state-the-union>.

You must resolve many issues for even the most straightforward crossword-puzzle-generator application. For example, how do you represent the grid of squares of a crossword puzzle inside the computer? Consider using a two-dimensional list where each element is one square. Some elements will be “black” and some “white.” Many “white” cells will include a number corresponding to a number in your across and down clues.

You need a source of words (i.e., a dictionary file) that can be read by the program. In what form should these words be stored to facilitate the complex manipulations required by the application?

You’ll want to print the clues containing definitions for each across and down word. Merely printing the blank puzzle is not a simple problem, especially if you’d like the black-squared regions to be symmetric as they often are in published crossword puzzles.

8.38 (Project: A Spell Checker) Many apps you use daily have built-in spell checkers. In this project, you’ll take a more mechanical approach. You’ll need a computerized dictionary as a source of words.

Why do we mistype so many words? Sometimes, we do not know the correct spelling, so we guess. In others, it’s because we transpose two letters (e.g., “defualt” instead of “default”). Sometimes we accidentally double-type a letter (e.g., “hanndy” instead of “handy”). Sometimes we type a nearby key instead of the one we intended (e.g., “birhyday” instead of “birthday”), and so on.

Design and implement a spell-checker application. Create a text file that has some words spelled correctly and some misspelled. Your program should look up each word in the dictionary. Your program should point out each incorrect word and suggest some correct alternatives that might have been what was intended.

For example, you can try all possible single transpositions of adjacent letters to discover that the word “defualt” is a direct match for “default.” Of course, this implies that your application will check all other single transpositions, such as “edfault,” “dfeault,” “deafult,” “defalut” and “default.” When you find a new transposition that matches a word in the dictionary, print it in a message, such as

Did you mean "default"?

Implement other tests, such as replacing each double letter with a single letter. Consider implementing other tests to improve your spell checker’s value.

8.39 (Project: Cooking with Healthier Ingredients) Write a program that helps users choose healthier ingredients when cooking and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, assume the recipe has no abbreviations for measures such as teaspoons, cups and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in the following table. Your program might display a warning such as, “Always consult your physician before making significant changes to your diet.”

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup margarine or yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)
1 cup milk	1 cup soy milk
1/4 cup oil	1/4 cup applesauce
white bread	whole-grain bread

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might use six egg whites instead. Conversion data for measurements and substitutes can be obtained at various websites. Your program should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

AI/Data-Science Case Study—Machine Learning with Simple Linear Regression

8.40 (*Project: Machine Learning with Simple Linear Regression: Statistics Can Be Deceiving*) Machine learning is one of the most exciting and promising subfields of artificial intelligence. Our goal here is to give you a friendly, hands-on introduction to one of the simpler machine-learning techniques.

Prediction

Machine learning is typically used to make predictions, based on existing data—and often lots of it. Wouldn't it be fantastic if you could improve weather forecasting to save lives, and minimize injuries and property damage? What if we could improve cancer diagnoses and treatment regimens to save lives, or improve business forecasts to maximize profits and secure people's jobs? What about detecting fraudulent credit-card purchases and insurance claims? How about predicting what prices houses are likely to sell for, ticket sales of new movies, and more generally, anticipated revenue of new products and services? How about predicting the best strategies for coaches and players to use to win more games and championships? All of these kinds of predictions are happening today with machine learning.

Boost Math Library and gnuplot

In this case study, you'll examine a completely coded program that demonstrates the machine-learning technique called **simple linear regression**, performed with a function from the open-source **Boost Math library**:

<https://github.com/boostorg/math>

which we also provide for your convenience in the `libraries` folder with the book's examples. This library defines many commonly used algorithms from engineering, science and mathematics. The program you'll study then passes commands to the 2D and 3D plotting application **gnuplot** to create several plot images. As you'll see, gnuplot uses its own plotting language different from C++, so in our code, we provide extensive comments explaining the gnuplot commands. To simplify, interacting with gnuplot, we use the free open-source library **gnuplot-cpp**,²⁵ which we also provide for your convenience in the `libraries` folder with the book's examples.

Descriptive Statistics

In data science, you'll often use statistics to describe and summarize your data. Some basic **descriptive statistics** are:

- **minimum**—the smallest value in a collection of values.
- **maximum**—the largest value in a collection of values.
- **range**—the difference between the maximum and minimum values.
- **count**—the number of values in a collection.
- **sum**—the total of the values in a collection.

Measures of dispersion (also called **measures of variability**), such as **range**, determine how spread out values are. Other measures of dispersion include **variance** and **standard deviation**.^{26,27,28}

Additional descriptive statistics include mean, median and mode. These are **measures of central tendency**. Each is a way of producing a single value representing a “central” value in a set of values—that is, one which is in some sense typical of the others. We discussed count, min, max, mean and median in Section 8.19.2.

Anscombe's Quartet

An important step in data analytics is “getting to know your data.” The **basic descriptive statistics** above certainly help you know more about your data. One caution, though, is that dramatically different datasets actually can have identical or nearly identical descriptive statistics. For an example of this phenomenon, consider **Anscombe's Quartet**:

https://en.wikipedia.org/wiki/Anscombe%27s_quartet

which consists of the following four sets of x - y coordinate pairs with 11 data samples each:

25. “gnuplot-cpp.” Accessed April 14, 2023. <https://github.com/martinruenz/gnuplot-cpp>.

26. “Understanding Descriptive Statistics.” Accessed April 14, 2023. <https://towardsdatascience.com/understanding-descriptive-statistics-c9c2b0641291>.

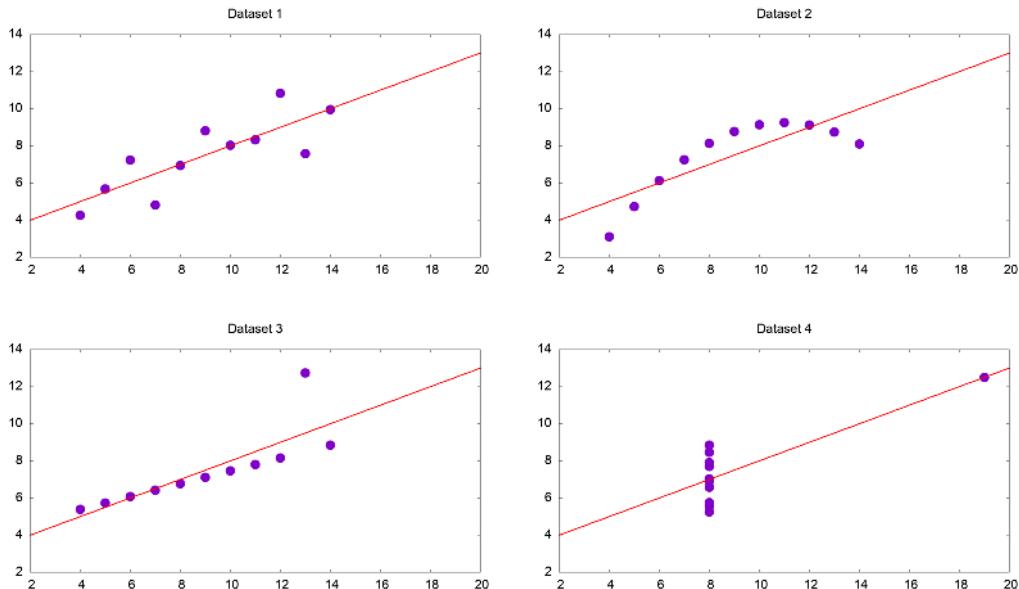
27. “Standard deviation.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Standard_deviation.

28. “Variance.” Accessed April 14, 2023. <https://en.wikipedia.org/wiki/Variance>.

$x1$	$y1$	$x2$	$y2$	$x3$	$y3$	$x4$	$y4$
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Interestingly, these datasets have several nearly identical descriptive statistics. For example, in all four datasets, the mean x -coordinate value is 9, and the mean y -coordinate value is 7.5.

The following diagrams—which our **fully coded case study** example creates—plot the $x1-y1$, $x2-y2$, $x3-y3$ and $x4-y4$ datasets, respectively:



We'll discuss the lines (known as **regression lines**) shortly. As you can see in the **visualizations**—but not necessarily by simply looking at the data in the preceding table—these Anscombe's Quartet datasets are **dramatically different**. Yet, like their descriptive statistics, their regression lines appear to be identical. This shows that you cannot draw conclusions only from descriptive statistics. You must use additional tools, like the visualizations above, to **get to know your data**.

Simple Linear Regression

Given a collection of points (x - y coordinate pairs) representing an **independent variable** (x) and a **dependent variable** (y), **simple linear regression** describes the **linear relationship** between the dependent and independent variables with a straight line, known as the **regression line**. The lines in the preceding diagrams are the regression lines for each of the four datasets in Anscombe's Quartet.

First, let's consider the linear relationship between Celsius and Fahrenheit temperatures. Given a Celsius temperature, we can calculate the corresponding Fahrenheit temperature using the following formula:

$$\text{fahrenheit} = 9 / 5 * \text{celsius} + 32$$

In this formula, `celsius` is the **independent variable**, and `fahrenheit` is the **dependent variable**. Each `fahrenheit` temperature **depends on** the `celsius` temperature used in the calculation. If we were to plot the Fahrenheit temperature for each Celsius temperature, all of the points would appear along the same straight line, revealing a **linear relationship** between the two temperature scales.

Components of the Simple-Linear-Regression Equation

The points along any straight line (in two dimensions)—like the regression lines shown in the preceding diagrams—can be calculated with the equation:

$$y = mx + b$$

where

- m is the line's **slope**,
- b is the line's **intercept** with the y -axis (at $x = 0$), or simply the **y -intercept**,
- x is the independent variable, and
- y is the dependent variable.

In the formula for converting Celsius temperatures to Fahrenheit temperatures:

- m is $9 / 5$,
- b is 32 ,
- x is `celsius`—the independent Celsius temperature, and
- y is `fahrenheit`—the dependent Fahrenheit temperature the calculation produces.

In simple linear regression, y is the **predicted value** for a given x . Of course, a line has an infinite number of points. If you can determine with simple linear regression the equation for a straight line from a modest finite number of sample points, you then have the means to make an infinite number of predictions, even for independent variable values you've never seen before.

How Simple Linear Regression Works

Simple linear regression is a **machine-learning** technique that determines the slope (m) and y intercept (b) of a straight line that “best fits” your data. The simple linear regression algorithm iteratively adjusts the slope and intercept and, for each adjustment, calculates the square of each point's distance from the line. The “best fit” occurs when the slope and intercept values minimize the sum of those squared distances. This is known as an **ordinary least squares** calculation.²⁹

Performing Simple Linear Regression with the Boost Math Library

The Boost Math Library's `simple_ordinary_least_squares` function encapsulates simple linear regression's calculations, giving you as results the slope and y -intercept for the straight line that best fits the data. After calling `simple_ordinary_least_squares`, you can plug into the $y = mx + b$ equation the slope (m) and intercept (b), then predict dependent y values, based on independent x values. We also use these values with `gnuplot` to display the regression line for the data along with the data points.

Comma-Separated Values (CSV) Files

We provided the Anscombe's Quartet data for you in the file `anscombe.csv`.³⁰ This file and this case-study exercise's source code are located in the `AnscombesQuartet` subfolder of this chapter's examples folder. The `.csv` filename extension indicates that the file is in **CSV (comma-separated values)** format—a particularly popular file format for distributing datasets. CSV files are simply text files in which each line is one record of information with its items separated by commas. The following are the first two rows of `anscombe.csv`:

```
x1,y1,x2,y2,x3,y3,x4,y4  
10,8.04,10,9.14,10,7.46,8,6.58
```

A CSV file's first row typically contains column names for the data in subsequent rows. In `anscombe.csv`, the remaining rows are the numeric values for the columns. Our code uses the `rapidcsv` techniques you learned in Section 8.19 to load the data into vectors.

Installing the GNU Scientific Library on macOS

On macOS, you can install the GNU Scientific Library using the `Homebrew` package manager³¹ as follows:

```
brew install gsl
```

Installing the GNU Scientific Library on Windows

In Visual Studio, you add the GNU Scientific Library to each project in which you wish to use it. With your project open in Visual Studio, perform the following steps:

1. Select Tools > NuGet Package Manager > Manage NuGet Packages for Solution....
2. In the Browse tab, search for "gsl-msvc-", then select `gsl-msvc-x64`.
3. In the right side of the NuGet package manager, click the checkbox next to your project's name, then click **Install** to add the library to your project.

Installing gnuplot on macOS

Install the gnuplot using the `Homebrew` package manager as follows:

```
brew install gnuplot
```

Installing gnuplot on Windows

Download and run the gnuplot Windows installer (`gp541-win64-mingw.exe`) from:

<https://sourceforge.net/projects/gnuplot/files/gnuplot/5.4.1/>

-
29. "Ordinary least squares." Accessed April 15, 2021. https://en.wikipedia.org/wiki/Ordinary_least_squares.
 30. In the Visual C++ version of this exercise's program, we assume `anscombe.csv` is in your user account's Documents folder. In the g++ and clang++ versions of this exercise's program, we assume `anscombe.csv` is in the same folder as the program.
 31. If the `brew` command is not found, visit <https://brew.sh/> for install instructions.

Click **Next >** until you reach the **Select Additional Tasks** step, then:

- Under **Select gnuplot's default terminal**, select the **windows** radio button.
- Scroll to the bottom of the settings and check the **Add application directory to your PATH environment variable** checkbox.

Click **Next >**, then **Install**. Once the installation completes, reboot your computer.

Compiling and Running the Program on Windows

Ensure that you've installed gnuplot. In your Visual Studio solution:

- Add to your project the file `anscombe_windows.cpp` from the `AnscombesQuartet` subfolder of this chapter's examples folder.
- Modify line 17 to specify the location of your user account's `Documents` folder.
- Ensure that your project's header-file include path contains the following folders from the `libraries` subfolder of this book's examples folder:

```
rapidcsv\src
gnuplot-cpp-windows\include
math-boost-1.81.0\include
```

- Build and run your project.

When you run the program it will create four PNG image files in your user account's `Documents` folder. Use **File Explorer** to navigate to that folder, then open the image files to view the four plots.

Compiling and Running the Program Using g++

First, install gnuplot in your Docker container by running the following two commands:

```
apt-get update -y
apt-get install -y gnuplot
```

To compile the program in the `g++` Docker container, change folders to the `AnscombesQuartet` subfolder of this chapter's examples folder, then execute:

```
g++ -std=c++20 -I ../../libraries/gnuplot-cpp/include/ \
-I ../../libraries/math-boost-1.81.0/include \
-I ../../libraries/rapidcsv/src anscombe_gnu.cpp
```

Compiling and Running the Program Using clang++

First, install gnuplot in your Docker container by running the following two commands:

```
apt-get update -y
apt-get install -y gnuplot
```

To compile the program in the `clang++` Docker container, change folders to the `AnscombesQuartet` subfolder of this chapter's examples folder, then execute:

```
clang++ -std=c++20 -I ../../libraries/fmt/include/ \
-I ../../libraries/math-boost-1.81.0/include \
-I ../../libraries/rapidcsv/src/ \
-I ../../libraries/gnuplot-cpp/include \
../../libraries/fmt/src/format.cc anscombe_clang.cpp
```

Extensively Commented Code

Next, study the code to learn how to use the `simple_ordinary_least_squares` function of the Boost Math Library to perform simple linear regression, and how to send gnuplot commands from a C++ program to the gnuplot application. Consider tweaking the gnuplot commands to see how your changes affect the plots our program produces. For example, you can change the plot's `pointtype`, `linewidth` and `linecolor` values.

AI/Data-Science Case Study—Machine Learning with Simple Linear Regression

8.41 (*Project: Machine Learning with Simple Linear Regression: Time Series Analysis*) Now that you've studied the code for Anscombe's Quartet, you can adapt it to other simple-linear-regression problems. Simple linear regression is commonly used to analyze **time series**—sequences of values (called **observations**) associated with points in time. Some examples are daily closing stock prices, hourly temperature readings, the changing positions of a plane in flight, annual crop yields and quarterly company profits. Perhaps the ultimate big-data time series is the stream of time-stamped tweets coming from Twitter users worldwide.

Time Series

For this exercise, you'll use simple linear regression to analyze a time series containing New York City's average January temperatures ordered by year for the years 1895–2023. This is a **univariate time series**—it contains **one observation per time**. A **multivariate time series** has **two or more observations** per time, such as hourly temperature, humidity and barometric-pressure readings in a weather application. Your goal in this exercise is to determine whether the regression line has:

- a negative slope, indicating a declining average temperature trend over that time,
- a zero slope, indicating a stable average temperature trend over that time, or
- a positive slope, indicating an increasing average temperature trend over that time.

Getting Weather Data from NOAA

The National Oceanic and Atmospheric Administration (NOAA)

<https://www.noaa.gov>

provides extensive public historical weather data, including time series for average temperatures in specific cities over various time intervals.

We obtained the New York City January average temperatures for 1895–2023 (the maximum date range available at the time of this writing) from NOAA's "Climate at a Glance" time series at:

<https://www.ncei.noaa.gov/access/monitoring/climate-at-a-glance/city/time-series>

You can select weather data for the entire U.S., regions within the U.S., states, cities and more. After selecting the data you need and the time frame to analyze, click **Plot** to display a diagram and view a table of the selected data. At the top of that table are icons you can click to download the data in several formats, including CSV.

For your convenience, we provided the file `nyc_ave_january_temps.csv` containing the data you'll use in this exercise. The file is located in the `nycdata` subfolder of this chap-

ter's examples folder. We also "cleaned" the data, so the file contains the following two columns per observation:

- Date—A value of the form YYYY (such as 2023). The downloaded data is in the form YYYYMM (such as 202301), where 01 represents January. We removed 01 from each data item in this column, leaving only the year.
- Temperature—A floating-point Fahrenheit temperature. We renamed this column from Value in the downloaded data.

We deleted a third column called Anomaly that's not required for this exercise.

Performing the Regression

Modify the Anscombe's Quartet code to perform simple linear regression using the New York City average January temperatures data and to plot the data with a regression line. What trend do you see over the last 129 years?



AI/Data Science Case Study: Intro to NLP—Who Wrote Shakespeare's Works?

8.42 (*Project: Intro to Similarity Detection with Very Basic Natural Language Processing*³²) Every day, we use **natural language** in various forms of communication, including:

- You read your text messages and check the latest news clips.
- You speak to family, friends and colleagues and listen to their responses.
- You have a deaf friend with whom you communicate via sign language and who enjoys close-captioned video programs.
- You have a blind colleague who reads braille, listens to audiobooks and listens to a screen reader speak about what's on the computer screen.
- You read e-mails, distinguishing junk from important communications.
- You receive a client e-mail in Spanish, translate it online, then respond in English, knowing that your client can easily translate your e-mail back to Spanish.
- You drive, observing road signs like "Stop," "Speed Limit 35" and "Road Under Construction."
- You give your car verbal commands like "call home" or "play classical music," or ask questions like, "Where's the nearest gas station?"
- You teach a child how to speak and read.
- You learn a foreign language.

Natural Language Processing (NLP) helps computers understand, analyze and process human text and speech. Natural language processing is performed on text collections composed of Tweets, Facebook posts, conversations, movie reviews, Shakespeare's plays, historical documents, news items, meeting logs, and so much more. A text collection is known as a **corpus**, the plural of which is **corpora**.

32. Much of this intro to natural language processing is borrowed from Chapter 12, Natural Language Processing, from our Pearson textbook *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud* (<https://deitel.com/pycds>).

Some common NLP applications include:

- **Natural language understanding**—understanding text or spoken language.
- **Sentiment analysis**—determining whether text has positive, neutral or negative sentiment. For example, companies analyze the sentiment of tweets about their products.
- **Readability assessment**—determining how readable text is, based on vocabulary, word lengths, sentence lengths, sentence structure, topics covered and more. While writing this book, we used the paid NLP tool Grammarly³³ to help us tune the writing to ensure the text's readability for a broad audience.
- **Intelligent virtual assistants**—software that helps you perform everyday tasks. Popular intelligent virtual assistants include Amazon Alexa, Apple Siri, Microsoft Cortana and Google Assistant.
- **Text summarization**—summarizing the key points of a large text. This can save valuable time for busy people.
- **Speech recognition**—converting speech to text.
- **Speech synthesis**—converting text to speech.
- **Language identification**—receiving a text when you don't know its language in advance, then automatically determining the language.
- **Interlanguage translation**—converting text to other spoken languages.
- **Named-entity recognition**—locating and categorizing items like dates, times, quantities, places, people, things, organizations and more.
- **Chatbots**—AI-based software that humans interact with via natural language. One popular chatbot application is automated customer support.
- **Similarity detection**—examining documents to determine how alike they are. Basic similarity metrics include average sentence length, frequency distribution of sentence lengths, average word length, frequency distribution of word lengths, frequency distribution of word usage, and more.

Many lower-level NLP tasks support the applications above as they perform their tasks, including:

- **Tokenization**—splitting text into **tokens**, which are meaningful units, such as words and numbers.
- **Parts-of-speech (POS) tagging**—identifying each word's part of speech, such as noun, verb, adjective, etc.
- **Noun phrase extraction**—locating groups of words representing nouns, such as “red brick factory.”³⁴
- **Spell checking and spelling correction.**

33. Grammarly also has a free version (<https://www.grammarly.com>).

34. The phrase “red brick factory” illustrates why natural language is such a difficult subject. Is a “red brick factory” a factory that makes red bricks? Is it a red factory that makes bricks of any color? Is it a factory built of red bricks that makes products of any type? In today’s music world, it could even a rock band’s name or the name of a game on your smartphone.

- **Stemming**—reducing words to their stems by removing prefixes or suffixes. For example, the stem of “varieties” is “variety.”
- **Lemmatization**—like stemming, but produces real words based on the original words’ context. For example, the lemmatized form of “varieties” is “variety.”
- **Word frequency counting**—determining how often each word appears in a corpus.
- **Stop-word elimination**—removing common words, such as *a, an, the, I, we, you* and more, to analyze the important words in a corpus.
- **n-grams**—producing sets of consecutive words in a corpus to identify words frequently appearing adjacent to one another. n-grams are commonly used for predictive text input, such as when your smartphone suggests possible next words as you type a text message.

This case study exercise serves two purposes:

- First, it introduces the crucial AI subtopic of natural language processing, which will play a key role in the future of anyone learning programming today.
- Second, it introduces the NLP subtopic of similarity detection, which you’ll perform using straightforward array-, string- and file-processing techniques.

Project Gutenberg

A great source of text for analysis is **Project Gutenberg**’s massive collection of free e-books:

<https://www.gutenberg.org>

The site contains over 70,000 e-books in various formats, including plain-text files. These are out of copyright in the United States. For information about Project Gutenberg’s Terms of Use and copyright in other countries, see:

https://gutenberg.org/policy/terms_of_use.html

For this case-study exercise, you’ll use the plain-text e-book files for William Shakespeare’s *Romeo and Juliet*:

<https://gutenberg.org/ebooks/1513>

and Christopher Marlowe’s *Edward the Second*:

<https://gutenberg.org/ebooks/20288>

Each of these is available free for download at Project Gutenberg.

Downloading E-Books from Project Gutenberg

Project Gutenberg does not allow programmatic access to its e-books. You must download the books to your own system before analyzing them.³⁵ To download *Romeo and Juliet* as a plain-text file, right-click the **Plain Text UTF-8** link on the play’s web page, then select

- **Save Link As...** (Chrome/Firefox/Microsoft Edge),
- **Download Linked File As...** (Safari), or

³⁵. “Information About Robot Access to our Pages.” Accessed April 14, 2023. https://www.gutenberg.org/policy/robot_access.html.

to save the play to the folder in which you'll place your solution to this exercise. Save the files with the names `RomeoAndJuliet.txt` and `EdwardTheSecond.txt`.

Who Wrote Shakespeare's Works?

Some people believe that William Shakespeare's works might have been penned by Christopher Marlowe, Sir Francis Bacon or others. You can learn more about this controversy at

https://en.wikipedia.org/wiki/Shakespeare_authorship_question

With some simple similarity-detection techniques, you can begin to compare Shakespeare's works with those of other authors. In this case-study exercise, your ultimate goal is to perform similarity detection between *Romeo and Juliet* and Christopher Marlowe's *Edward the Second* to determine whether Christopher Marlowe might have authored Shakespeare's works. We explore more sophisticated similarity detection techniques in the "Natural Language Processing" chapter of our Pearson Education textbook *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud* (<https://deitel.com/pycds>).

Analyzing Romeo and Juliet to Prepare for Simple Similarity Detection

You'll now perform some simple statistical analysis to determine document similarity. Begin by focusing on Shakespeare's *Romeo and Juliet*. Later, you'll perform the same tasks on *Edward the Second*, then compare your analyses' results. As a control, you could also analyze a play from a third author who is not involved in this controversy. As your program reads and processes *Romeo and Juliet*, it should keep track of the following items, then use them to display various statistics:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The number of sentences of each length.
- The number of words of each length.
- The unique words' frequencies.

Cleaning Romeo and Juliet Before Analyzing It

Data does not always come ready for analysis. It could, for example, be in the wrong format. Data scientists spend much of their time preparing data before performing analyses. Preparing data for analysis is called **data munging** or **data wrangling**.

Each e-book you download from Project Gutenberg contains information and legal paragraphs that you will not want to include in your analyses. You should open *Romeo and Juliet* in a text editor and "clean" it by removing the Project Gutenberg text. Remove everything from the beginning of the document up to and including the title "THE TRAGEDY OF ROMEO AND JULIET," then remove everything from the following text through the end of the file:

End of the Project Gutenberg EBook of Romeo and Juliet,
by William Shakespeare

You should do some additional text cleaning with a text editor before running your analytics on the play:

- Each character’s name is mentioned each time that character speaks—this is standard in plays. You don’t need the characters’ names for the particular analytics you’ll run in this case study. In fact, they’ll “get in the way.” For more sophisticated similarity detection, you might want to keep them.
- Plays also include staging directions indicating when characters should enter and leave the stage, duel one another, fall down and die when poisoned, and the like. These directions also should be removed.

You could write a program to handle these cleaning chores. Be careful, though—scrutinizing the manuscript may reveal many special cases your code needs to handle. Programming for them could be time-consuming and error-prone.

Vectors You’ll Need to Perform Your Analysis

You are now ready to analyze *Romeo and Juliet* to create the statistics you’ll use for simple similarity detection. Use four vectors to record various counts that characterize the play’s text.

- **sentenceLengths** will keep counts of how many sentences consist of one word, two words, three words, etc.
- **wordLengths** will keep counts of how many words consist of one character, two characters, three characters, etc.
- **words** will contain each unique word in the play.
- **frequencies** will contain the count of how many times the word at the corresponding index of **words** appears in the play.

Analyzing a Sentence

Consider:

*“O Romeo, Romeo! Wherefore art thou Romeo?”*³⁶

- There are seven words, so your program would add 1 to **sentenceLengths[7]**.
- The first word (“O”) has one letter, so your program would add 1 to **wordLengths[1]**.
- The second word (“Romeo”) has five letters, so your program would add 1 to **wordLengths[5]**, and so on.

To perform word-frequency counting, convert the words to lowercase letters so that all occurrences of the same word will compare as equal. As you process each word, search the **words** vector to determine whether the word is already in the vector. If so, add one to that word’s count at the corresponding index in **frequencies**. Otherwise, insert the word in alphabetical order in **words** and insert the count 1 at the same index in **frequencies**.

For example, after converting the text to all lowercase letters, when the program encounters the first occurrence of “romeo” in

“o romeo, romeo! wherefore art thou romeo?”

it will insert “romeo” in alphabetical order in **words** and insert 1 at the same location in **frequencies**. When the program encounters each subsequent occurrence of “romeo,” it

36. Shakespeare, William. “Romeo and Juliet,” Act II, Scene II. Accessed April 14, 2023. http://shakespeare.mit.edu/romeo_juliet/full.html.

will find the index of “romeo” in `words`, then increment the corresponding `frequencies` element.

Implementing Your Analysis Code

Use the string- and file-processing techniques you’ve learned to read the contents of *Romeo and Juliet* and perform the following tasks:

- Every sentence ends with a **sentence terminator**—a period (.), a **question mark** (?) or an **exclamation point** (!). Define a `processSentence` function that reads words until it encounters a sentence terminator. This function updates the sentence, word and character counters as you process each word. When you hit the end of a sentence, increment the appropriate counter in the `sentenceLengths` vector and reset the word counter to zero.
- For every word, `processSentence` should call the `processWord` function to search the `words` vector for that word and either increment the word’s counter at the corresponding index in `frequencies` or insert the word in alphabetical order in `words` and insert the count 1 at the corresponding index in `frequencies`.

Remember to keep track of the total number of sentences, words and characters.

Analysis Report

Next, display the following statistics for *Romeo and Juliet*:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The mean (average) sentence length.
- The mean word length.
- The median sentence length.
- The median word length.
- A table of sentence lengths and their percentages among all sentence lengths.
- A table of word lengths and their percentages among all word lengths.
- A frequency-distribution table containing the play’s unique words, their frequencies and their percentages among all words in the play. Display these in descending order by frequency.

Your program also should output these statistics to a file to make it easier to study the results you produce and compare them between plays.

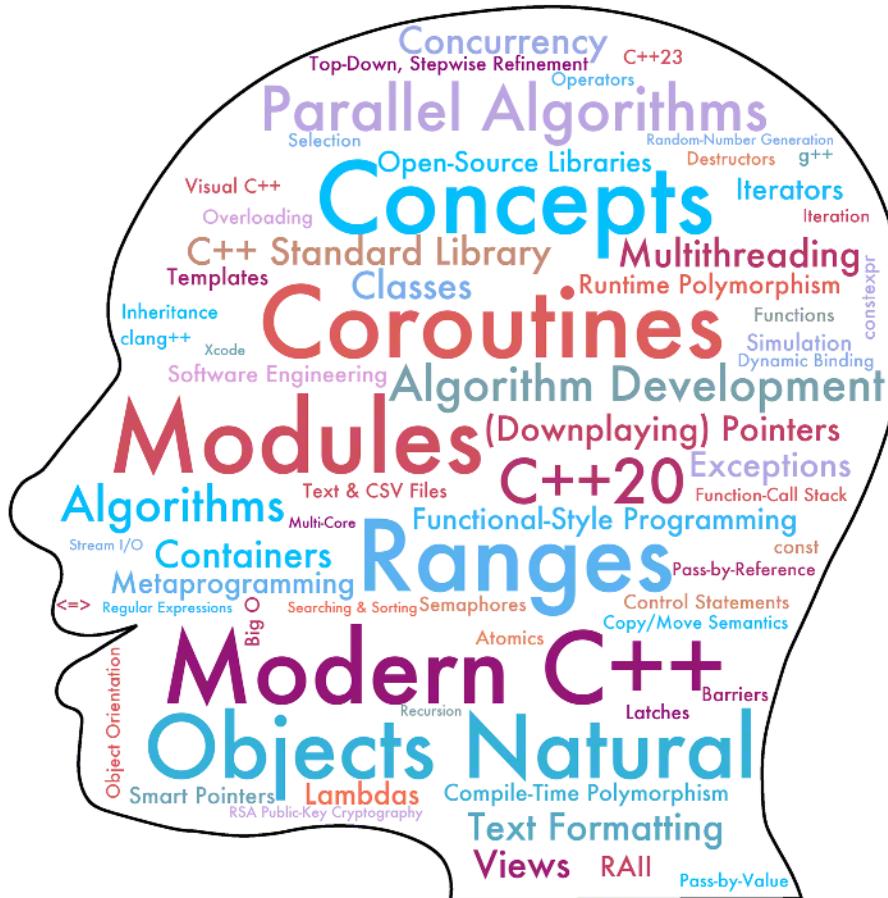
Analyzing Christopher Marlowe’s Play *Edward the Second*

Once you’ve analyzed *Romeo and Juliet*, use a text editor to clean Christopher Marlowe’s play *Edward the Second*. As part of any data-science study, it’s important to get to know your data. The conventions used in *Edward the Second* for specifying who’s speaking and the play’s staging directions differ from those you saw in *Romeo and Juliet*. So, be careful to observe these differences when cleaning *Edward the Second*. After you clean the text, run your analytics program on *Edward the Second*. Compare the analytics with those you produced for *Romeo and Juliet*. Comment on the similarities you find between these plays.

This page intentionally left blank

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.
 - Enforce data encapsulation of **private** data members using **public** *get* and *set* functions.
 - Use a constructor to initialize an object's data.
 - Separate a class's interface from its implementation for reuse.
 - Access class members via dot (.) and arrow (->) operators.
 - Use destructors to perform “termination housekeeping” as objects go out of scope.
 - 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.
 - Create aggregate types via **structs** and use C++20 designated initializers to initialize aggregate members.
 - Conclude our Objects-Natural track by studying the Vigenère secret-key cipher implementation used in an earlier case study.

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 <code>private</code> and <code>public</code>	
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 (<code>::</code>)	
9.7.7	Member Function <code>setTime</code> and Throwing Exceptions	
9.7.8	Member Functions <code>to24HourString</code> and <code>to12HourString</code>	
9.7.9	Implicitly Inlining Member Functions	
9.7.10	Member Functions vs. Global Functions	
9.7.11	Using Class Time	
9.7.12	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 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 <code>private</code> Data Member	
9.15	Default Assignment Operator	
9.16	<code>const</code> Objects and <code>const</code> Member Functions	
9.17	Composition: Objects as Members of Classes	
9.18	<code>friend</code> Functions and <code>friend</code> Classes	
9.19	The <code>this</code> Pointer	
9.19.1	Implicitly and Explicitly Using the <code>this</code> Pointer to Access an Object's Data Members	
9.19.2	Using the <code>this</code> 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	Concluding Our Objects Natural Case Study Track: Studying the Vigenère Secret-Key Cipher Implementation	
9.23	Wrap-Up	
	Exercises	

9.1 Introduction¹

Section 1.9 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 perform powerful tasks without knowing 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 define becomes a new type you can use to create objects. Some development teams in

1. This chapter depends on the terminology and concepts introduced in Section 1.9, Introduction to Object Orientation.

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 objects of 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 your career, 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.

```
1 // Fig. 9.1: AccountTest.cpp
2 // Creating and manipulating an Account object.
3 #include <iostream>
4 #include <string>
5 #include "Account.h"
6
7 int main() {
8     Account myAccount{}; // create Account object myAccount
9
10    // show that the initial value of myAccount's name is the empty string
11    std::cout << std::format("Initial account name: {}\n",
12                           myAccount.getName());
13
14    // prompt for and read the name
15    std::cout << "Enter the account name: ";
16    std::string name{};
17    std::getline(std::cin, name); // read a line of text
18    myAccount.setName(name); // put name in the myAccount object
19
20    // display the name stored in object myAccount
21    std::cout << std::format("Updated account name: {}\n",
22                           myAccount.getName());
23
24 }
```

```
Initial account name:
Enter the account name: Olga Novikova
Updated account name: Olga Novikova
```

Fig. 9.1 | Creating and manipulating an Account object.

Instantiating an Object

Typically, you cannot call a class's member functions until you create an object of that class.² Line 9

```
Account myAccount{}; // create Account object myAccount
```

2. In Section 9.20, you'll see that `static` member functions can be called without creating objects.

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—a fundamental type built into C++. In line 9, 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 with C++ standard library and third-party library classes throughout this book.

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 the class `Account` and 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.⁴

Calling Class `Account`'s `getName` Member Function

Class `Account`'s `getName` member function returns the name stored in a particular `Account` object. Line 13 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 specific `Account` object. Line 19 calls `myAccount.setName` member function to store `name`'s value in `myAccount`.

Displaying the Name That Was Entered by the User

To confirm that `myAccount` now contains the name you entered, line 23 calls `myAccount.getName` member function again and displays its result.



Checkpoint

- 1 *(Fill-in)* When you `#include` a header defined in your program, enclose it in _____ rather than angle brackets (<>).

Answer: double quotes ("").

- 2 *(Code)* Assume your program contains a custom class `Employee`. Write a preprocessor directive that includes the class's definition from the header `Employee.h`.

Answer: `#include "Employee.h"`

-
3. C++ standard library headers, like `<iostream>`, do not use the `.h` filename extension and some C++ programmers prefer the `.hpp` extension.
 4. Even your custom class headers can be `#included` by placing them in angle brackets (< and >) by specifying your program's folder as part of the compiler's header search path, as you've done for third-party libraries in the Objects-Natural case studies.

9.3 Account Class with a Data Member and Set and Get Member Functions

Now that we've seen the `Account` class in action (Fig. 9.1), we present its internal details.

9.3.1 Class Definition

Class `Account` (Fig. 9.2) contains the data member `m_name` (line 19) to store the account holder's name. Each object of a class has its own copy of the class's data members.⁵ In Section 9.6, we'll add a `balance` data member to keep track of each `Account`'s balance. Class `Account` also contains member functions:

- `setName` (lines 10–12) stores a name in an `Account`, and
- `getName` (lines 15–17) retrieves a name from an `Account`.

```

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

```

Fig. 9.2 | Account class with a data member and member functions to set and get its value.

Keyword `class` and the Class Body

The class definition begins with the keyword `class` (line 7), followed immediately by the class's name (`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, and each subsequent word begins with a capital first letter.

Every class's body is enclosed in braces {} (lines 7 and 20). The class definition terminates with a required semicolon (line 20).

5. In Section 9.20, you'll see that `static` data members are an exception.

Data Member `m_name` of Type `std::string`

Recall from Section 1.9 that an object has attributes. These are implemented as data members. Each object maintains its own copy of these throughout its lifetime—that is, while the object exists in memory. Usually, a class also contains member functions that you can use to manipulate particular objects’ data members.

Data members are declared inside a class definition but outside the class’s member functions. Line 19

```
std::string m_name; // data member containing account holder's 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 13 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 `Account.h` (Fig. 9.2), we use `std::` when referring to `string_view` (line 10) and `string` (lines 15 and 19). **Headers should not contain global scope `using` directives or `using` declarations.** These would be `#included` in other source files, potentially causing naming conflicts. Since Chapter 6, we’ve qualified every standard library class name, function name and object (e.g., `std::cout`) with `std::` as a good practice.

`setName` Member Function

Member function `setName` (lines 10–12) 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. Line 11 copies the `name` parameter’s `string` 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) 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 `getName`’s parameter list (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 in this member function that would modify the object. It also tells the compiler that `getName` may be called on a `const` `Account` object or via a reference or pointer to a `const Account`.





Checkpoint

1 (*True/False*) All objects of one class share one copy of the class's data members.

Answer: False. Actually, each object of a class has its own copy of the class's data members.

2 (*Fill-in*) Declaring a member function _____ tells the compiler, "this function should not modify the object on which it's called."

Answer: const.

3 (*Code*) Write two `string` declarations that could be used in class `Account` to specify that each object of the class has a first name and a last name.

Answer: `std::string m(firstName);`
`std::string m(lastName);`

9.3.2 Access Specifiers `private` and `public`

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.⁶ This is known as **information hiding** (or hiding implementation details) and is a recommended practice of the C++ Core Guidelines⁷ and object-oriented programming in general. 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 access specifier `private`.⁸

CG

This class also contains the `public` access specifier (line 8):

```
public:
```

Class members listed after `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 Section 9.7, we’ll say more about `private`’s benefits. In Chapter 10, we’ll introduce the access specifier `protected`.

AS

Default Access for Class Members

By default, class members are `private` unless you specify otherwise. After you list an access specifier, every subsequent class member has that access level until you list a different access specifier. The `public` and `private` access specifiers may be repeated, but this can be confusing. We prefer to group all of the class’s `public` members under one `public` access specifier and group all of the class’s `private` members under one `private` access specifier.

6. Or to “friends” of the class, as you’ll see in Section 9.18.

7. C++ Core Guidelines, “C.9: Minimize Exposure of Members.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-private>.

8. We generally omit the colon when referring to `private` and `public` in sentences, as in this footnote.



Checkpoint

- 1 (*Fill-in*) Data member declarations that appear after `private`: indicate the declarations are accessible only to that class's member functions. This is known as _____ and is a recommended practice of the C++ Core Guidelines in particular and object-oriented programming in general.

Answer: information hiding.

- 2 (*True/False*) By default, class members are `public` unless you specify otherwise.

Answer: False. Actually, by default, class members are `private` unless you specify otherwise.

- 3 (*True/False*) After you list an access specifier, every subsequent class member has that access level.

Answer: False. Actually, after you list an access specifier, every subsequent class member has that access level *until* you list a different access specifier.

9.4 Account Class: Custom Constructors

As you saw in Section 9.3, creating an `Account` object initializes its `string` data member `m_name` to the empty `string` by default. But what if you want to provide a name when creating 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 a constructor call when you create an object of a class type, 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 10 of Fig. 9.4:

```
Account account1{"Mia Gonzalez"};
```

Here, the "Mia Gonzalez" is passed to the `Account` class's constructor 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 defines the `Account` class with a constructor that receives a `name` parameter and uses it to initialize the data member `m_name` when an `Account` object is created.

```
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
```

Fig. 9.3 | Account class with a constructor that initializes the account name. (Part 1 of 2.)

```

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;
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
25 }; // end class Account

```

Fig. 9.3 | Account class with a constructor that initializes the account name. (Part 2 of 2.)

Account Class's Custom Constructor Definition

Lines 9–12 of Fig. 9.3 define Account's constructor. Usually, constructors are `public`, so any code with access to the class definition can create and initialize objects of that class. Line 9 indicates that the constructor has a `string_view` parameter. When creating a new `Account` object, you must pass a person's name to the constructor's `string_view` parameter. The constructor then initializes the data member `m_name` with the parameter's contents. Line 9 of Fig. 9.3 does not specify a return type, not even `void`, because constructors cannot return values. Also, constructors cannot be declared `const`—initializing an object must modify it.

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 (`:`).

Each member initializer consists of a data member's variable name followed by braces containing its initial value.⁹ 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 constructors execute in the order you declare the data members in the class. For clarity, list the member initializers in the same order—some compilers will warn you if you do not. The member-initializer list executes before the constructor's body.



9. 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.

 Though you can perform initialization with assignment statements in the constructor's body, the C++ Core Guidelines recommend using member initializers.¹⁰ 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 values to them in the constructor's body.

 Declare a constructor **explicit** if it can be called with one argument—that is, it has one parameter or it has additional parameters with default arguments. This prevents the compiler from using the constructor to perform implicit type conversions.¹¹ The keyword **explicit** means that `Account`'s constructor must be called explicitly, as in

```
Account account1{"Mia Gonzalez"};
```

 For now, simply declare all single-parameter constructors **explicit**. In Section 11.9, 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.

Initializing Account Objects When They're Created

The program in Fig. 9.4 initializes two `Account` objects using the constructor. Line 10 creates the `Account` object named `account1`:

```
Account account1{"Mia Gonzalez"};
```

This calls the `Account` constructor (lines 9–12 of Fig. 9.3), which uses the argument "Mia Gonzalez" to initialize the new object's `m_name` data member. Line 11 of Fig. 9.4 repeats this process, passing the argument "Asahi Susuki" to initialize `m_name` for the object `account2`:

```
Account account2{"Asahi Susuki"};
```

To confirm the objects were initialized properly, lines 14–16 call the `getName` member function on each `Account` object to get its name. The output shows that each `Account` has a different name, confirming that each object has its own copy of data member `m_name`.

```

1 // Fig. 9.4: AccountTest.cpp
2 // Using the Account constructor to initialize the m_name
3 // data member when each Account object is created.
4 #include <iostream>
5 #include <iomanip>
6 #include "Account.h"
7

```

Fig. 9.4 | Using the `Account` constructor to initialize the `m_name` data member at the time each `Account` object is created. (Part 1 of 2.)

-
10. C++ Core Guidelines, “C.49: Prefer Initialization to Assignment in Constructors.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-initialize>.
 11. C++ Core Guidelines, “C.46: By Default, Declare Single-Argument Constructors **explicit**.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-explicit>.

```
8 int main() {
9     // create two Account objects
10    Account account1{"Mia Gonzalez"};
11    Account account2{"Asahi Susuki"};
12
13    // display each Account's corresponding name
14    std::cout << std::format(
15        "account1 name is: {}\naccount2 name is: {}",
16        account1.getName(), account2.getName());
17 }
```

```
account1 name is: Mia Gonzalez
account2 name is: Asahi Susuki
```

Fig. 9.4 | Using the `Account` constructor to initialize the `m_name` data member at the time each `Account` object is created. (Part 2 of 2.)

Default Constructor

Recall that line 9 of Fig. 9.1 created an `Account` object with empty braces to the right of the object's variable name:

```
Account myAccount{};
```

In this statement, C++ implicitly calls `Account`'s **default constructor**. The compiler generates a default constructor with no parameters in any class that does not define a constructor. 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 ("").

When you declare an object with empty braces, as in the preceding statement, data members that are not explicitly initialized get value initialized (introduced in Section 6.5). Without the braces, such data members contain undefined (“garbage”) values.

There's No Default Constructor in a Class That Defines a Constructor

The compiler will not create a default constructor for a class if you provide a **custom constructor**. 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, you should initialize them in their declarations or provide a custom constructor that initializes them 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—as well as C++20's new three-way comparison operator. This chapter briefly introduces copy construction, copy

assignment and destructors. Chapter 11 introduces the three-way comparison operator and discusses the details of all these special member functions, including

- 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 design your classes so the compiler can auto-generate the special member functions for you. This is called the "Rule of Zero," meaning that you provide no special member functions in the class's definition.



Checkpoint

1 (*Discussion*) Why are constructors usually `public`?

Answer: So any code with access to the class definition can create and initialize objects of that class.

2 (*Discussion*) In any class that does not define a constructor, the compiler generates a default constructor with no parameters. What does that default constructor do?

Answer: 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.

3 (*Code*) Assume class `Account` has two data members named `m(firstName` and `m.lastName`. Define a constructor that would initialize both data members using strings passed to the constructor. This constructor must be called with two arguments, so you do not need to declare it `explicit`.

Answer:

```
Account(std::string_view firstName, std::string_view lastName)
: m(firstName{firstName}, m.lastName{lastName} {
    // empty body
}
```

9.5 Software Engineering with Set and Get Member Functions

As you'll see in the next section, *set* member functions can validate attempts to modify `private` data, and *get* member functions can control how that data is presented to the caller. 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 do whatever it wants with the variable by calling `public` *set* and *get* functions. You'd think that you could call the `public` *get* function to peek at the `private` data and see exactly how it's stored in the object, and that you could call the `public` *set* function to modify the `private` data at will.



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,

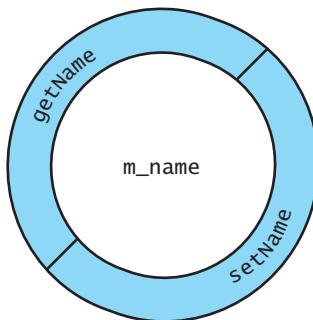
- a day in March outside the range 1 through 31 or
 - a product code not in the company's product catalog.

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` in the inner circle is hidden inside the object and accessible only via an outer layer of **public** member functions, represented by the shaded outer ring containing `getName` and `setName`. Any client that needs to interact with the Account object can only call the **public** member functions of the outer layer.



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—possibly often.



Checkpoint

- (Fill-in)** set member functions can validate attempts to modify private data, and member functions can control how that data is presented to the caller.

Answer: *get.*

- 2** (*True/False*) Even though a class's clients cannot directly access the class's private data members, the client can nevertheless do whatever it wants with that variable through `public set` and `get` functions.

Answer: False. Actually, *set* functions can be written to validate their arguments and reject any attempts to *set* the data to incorrect values. A *get* function can present the data in a different form, keeping the object's actual data representation hidden from the user.

- 3** (*True/False*) Generally, data members are **private**, and the member functions that a class's clients need to use are **public**.

Answer: True.

9.6 Account Class with a Balance

Figure 9.5 defines an Account class that maintains two related pieces of data—the account holder’s name and bank balance. The C++ Core Guidelines recommend defining related data items in a class (or in a struct, as you’ll see in Section 9.21).¹²

```

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 <algorithm>
5 #include <string>
6 #include <string_view>
7
8 class Account {
9 public:
10    // Account constructor with two parameters
11    Account(std::string_view name, double balance)
12        : m_name{name}, m_balance{std::max(0.0, balance)} { // member init
13            // empty body
14    }
15
16    // function that deposits (adds) only a valid amount to the balance
17    void deposit(double amount) {
18        if (amount > 0.0) { // if the amount is valid
19            m_balance += amount; // add it to m_balance
20        }
21    }
22
23    // function that returns the account balance
24    double getBalance() const {
25        return m_balance;
26    }
27
28    // function that sets the account name
29    void setName(std::string_view name) {
30        m_name = name; // replace m_name's value with name
31    }
32
33    // function that returns the account name
34    const std::string& getName() const {
35        return m_name;
36    }
37 private:
38    std::string m_name;
39    double m_balance;
40}; // 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.

12. C++ Core Guidelines, “C.1: Organize Related Data into Structures (structs or classes).” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-org>.

Data Member `balance`

A bank services many accounts, each with its own balance. Every `Account` object now has its own `m_name` and `m_balance`. Line 39 declares a `double` data member `m_balance`.¹³

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 11–14) receives a second parameter `balance` (a `double`) representing the starting balance. We did not declare this constructor `explicit` because it cannot be called with only one parameter.

The `m_balance` member initializer calls the `std::max` function (header `<algorithm>`) to initialize `m_balance` to `0.0` or `balance`, whichever is greater. This ensures `m_balance` has a valid non-negative value when creating an `Account` object. In Section 9.7, we'll validate arguments in the constructor body and use exceptions to indicate invalid arguments.

`deposit` Member Function

Member function `deposit` (lines 17–21) receives a `double` parameter `amount` and does not return a value. Lines 18–20 ensure that parameter `amount`'s value is added to `m_balance` only if `amount` is a valid deposit amount greater than zero.

`getBalance` Member Function

Member function `getBalance` (lines 24–26) allows the class's clients to obtain an `Account` object's `m_balance` value. The member function specifies the return type `double` and an empty parameter list. It is declared `const` because returning `m_balance`'s value 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 8–9). It initializes them with a valid balance of `50.00` and an invalid balance of `-7.00`, respectively. Lines 12–15 output both Accounts' names and balances by calling each object's `getName` and `getBalance` member functions. Our class ensures that initial balances are greater than or equal to zero. The `account2` object's `balance` is initially `0.0` because the constructor rejected the attempt to start `account2` with a negative balance by setting its value to `0.0`.

13. As a reminder, industrial-strength financial applications should not use `double` to represent monetary amounts.

```

1 // Fig. 9.6: AccountTest.cpp
2 // Displaying and updating Account balances.
3 #include <iostream>
4 #include <iomanip>
5 #include "Account.h"
6
7 int main() {
8     Account account1{"Mia Gonzalez", 50.00};
9     Account account2{"Asahi Susuki", -7.00};
10
11    // display initial balance of each object
12    std::cout << std::format("account1: {} balance is ${:.2f}\n",
13                           account1.getName(), account1.getBalance());
14    std::cout << std::format("account2: {} balance is ${:.2f}\n\n",
15                           account2.getName(), account2.getBalance());
16

```

```

account1: Mia Gonzalez balance is $50.00
account2: Asahi Susuki balance is $0.00

```

Fig. 9.6 | Displaying and updating Account balances.

Reading a Deposit Amount from the User and Making a Deposit

Lines 17–21 prompt for, input and display the account1 deposit amount. Line 22 calls object account1's deposit member function, passing the amount to add to account1's balance. Lines 25–28 output both Accounts' names and balances again to show that only account1's balance changed.

```

17 std::cout << "Enter deposit amount for account1: " // prompt
18 double amount;
19 std::cin >> amount; // obtain user input
20 std::cout << std::format(
21     "adding ${:.2f} to account1 balance\n\n", amount);
22 account1.deposit(amount); // add to account1's balance
23
24 // display balances
25 std::cout << std::format("account1: {} balance is ${:.2f}\n",
26                           account1.getName(), account1.getBalance());
27 std::cout << std::format("account2: {} balance is ${:.2f}\n\n",
28                           account2.getName(), account2.getBalance());
29

```

```

Enter deposit amount for account1: 25.37
adding $25.37 to account1 balance

```

```

account1: Mia Gonzalez balance is $75.37
account2: Asahi Susuki balance is $0.00

```

Lines 30–33 prompt for, input and display account2's deposit amount. Line 34 calls object account2's deposit member function with the argument amount 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 changed.

```
30 std::cout << "Enter deposit amount for account2: " // prompt
31 std::cin >> amount; // obtain user input
32 std::cout << std::format(
33     "adding ${:.2f} to account2 balance\n\n", amount);
34 account2.deposit(amount); // add to account2 balance
35
36 // display balances
37 std::cout << std::format("account1: {} balance is ${:.2f}\n",
38     account1.getName(), account1.getBalance());
39 std::cout << std::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: Mia Gonzalez balance is $75.37
account2: Asahi Susuki balance is $123.45
```



Checkpoint

- I (*Fill-in*) The _____ standard library function returns the greater of its two arguments.

Answer: `std::max`.

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.

Conventional software-engineering wisdom says that to use an object of a class, the  client code (e.g., `main`) needs to know only

- which member functions to call,
- which arguments to provide to each member function and
- which 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. **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:

- Separating interface from implementation, and
- using the preprocessor directive “`#pragma once`” in a header to prevent the header code from being included in the same source-code file more than once—a class can be defined only once, so this prevents multiple-definition errors.

C++20 Modules Change How You Separate Interface from Implementation



As you’ll see in Chapter 16, C++20 modules¹⁵ eliminate the need for preprocessor directives like `#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.

9.7.1 Interface of a Class

Interfaces define and standardize how people and systems interact with one another. For example, a radio’s controls serve as an interface between users and its internal components. The controls allow users to perform specific 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 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) defines class `Time` and `Time.cpp` (Fig. 9.8) defines class `Time`’s member functions. This split

- helps make the class reusable,
- 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
- enables the clients to ignore how the class’s member functions are implemented.



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 uses objects of our `Time` class.

14. Rather than building your own classes to represent times and dates, you’ll typically use capabilities from the C++ standard library header `<chrono>` (<https://en.cppreference.com/w/cpp/chrono>).

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.

9.7.3 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.

```

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 };

```

Fig. 9.7 | Time class definition.

The header still specifies the class's `private` data members (lines 14–16). Each uses an **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.

The C++ Core Guidelines recommend using in-class initializers when a data member should be initialized with a constant.¹⁶ The Core Guidelines also recommend that, when possible, you initialize all your data members with in-class initializers and let the compiler generate a default constructor for your class—this constructor can be more efficient than one you define.¹⁷



#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

16. C++ Core Guidelines, “C.48: Prefer In-Class Initializers to Member Initializers in Constructors for Constant Initializers.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-in-class-initializer>.
17. C++ Core Guidelines, “C.45: Don’t Define a Default Constructor That Only Initializes Data Members; Use In-Class Member Initializers Instead.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default>.

 headers that might 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 in the same source-code file more than once. This is sometimes referred to as an **include guard**. In Chapter 16, 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 23 and 28).

```

1 // Fig. 9.8: Time.cpp
2 // Time class member-function definitions.
3 #include <format>
4 #include <stdexcept> // for invalid_argument exception class
5 #include <string>
6 #include "Time.h" // include definition of class Time from Time.h
7
8 // set new Time value using 24-hour time
9 void Time::setTime(int hour, int minute, int second) {
10     // validate hour, minute and second
11     if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
12         (second < 0 || second >= 60)) {
13         throw std::invalid_argument{
14             "hour, minute or second was out of range"};
15     }
16
17     m_hour = hour;
18     m_minute = minute;
19     m_second = second;
20 }
21
22 // return Time as a string in 24-hour format (HH:MM:SS)
23 std::string Time::to24HourString() const {
24     return std::format("{:02d}::{:02d}::{:02d}", m_hour, m_minute, m_second);
25 }
26
27 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
28 std::string Time::to12HourString() const {
29     return std::format("{}:{}{:02d}::{:02d} {}",
30                         ((m_hour % 12 == 0) ? 12 : m_hour % 12), m_minute, m_second,
31                         (m_hour < 12 ? "AM" : "PM"));
32 }
```

Fig. 9.8 | Time class member-function definitions.

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 9, 23 and 28). 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 9, 23 and 28) 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 or call the class's member functions without receiving a `Time` object or a pointer or reference to one. 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 17–19, 24 and 30–31 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 9–20) is a **public** function that declares three `int` parameters and uses them to set the time. Lines 11–12 test each argument to determine whether the value is in range. If so, lines 17–19 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 13–14) of type **invalid_argument** (header `<stdexcept>`), notifying 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, initializing it with 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—this 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 Functions `to24HourString` and `to12HourString`

Member function `to24HourString` (lines 23–25 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 24 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.

Function `to12HourString` (lines 28–32) 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 30 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 31 determines whether to include AM or PM.

9.7.9 Implicitly Inlining Member Functions

 If a member function is fully defined in a class's body (as in our `Account` class examples), it's implicitly declared `inline` (Section 5.11). 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, provided that the compiler has access to the function's definition.

 Only the simplest, most stable member functions (i.e., whose implementations are unlikely to change) and those that are the most performance-sensitive 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.

9.7.10 Member Functions vs. Global Functions

 Member functions `to24HourString` and `to12HourString` take no arguments. They implicitly know about and can access the data members for the `Time` object on which they're invoked. This is a 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.

9.7.11 Using Class Time

Once class `Time` is defined, it can be used as a type in declarations, as in:

```
Time sunset{}; // object of type Time
std::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
```

Figure 9.9 creates and manipulates a `Time` object. Separating `Time`'s interface from the implementation of its member functions does not affect how 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 17) and can ensure that `Time` objects are created and manipulated correctly in the client code.

```
1 // fig09_09.cpp
2 // Program to test class Time.
3 // NOTE: This file must be linked with Time.cpp.
4 #include <format>
5 #include <iostream>
6 #include <stdexcept> // invalid_argument exception class
7 #include <string_view>
8 #include "Time.h" // definition of class Time from Time.h
9
10 // displays a Time in 24-hour and 12-hour formats
11 void displayTime(std::string_view message, const Time& time) {
12     std::cout << std::format("{}\n24-hour time: {}\n12-hour time: {}{}\n",
13         message, time.to24HourString(), time.to12HourString());
14 }
15
16 int main() {
17     Time t{}; // instantiate object t of class Time
18
19     displayTime("Initial time:", t); // display t's initial value
20     t.setTime(13, 27, 6); // change time
21     displayTime("After setTime:", t); // display t's new value
22
23     // attempt to set the time with invalid values
24     try {
25         t.setTime(99, 99, 99); // all values out of range
26     }
27     catch (const std::invalid_argument& e) {
28         std::cout << std::format("Exception: {}{}\n", e.what());
29     }
30
31     // display t's value after attempting to set an invalid time
32     displayTime("After attempting to set an invalid time:", t);
33 }
```

```
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.

Throughout the program, we display the `Time` object's `string` representations using the `displayTime` function (lines 11–14), which calls `Time`'s `to24HourString` and `to12HourString` member functions. Line 17 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 19 displays the time in 24-hour and 12-hour formats to confirm that the members were correctly initialized. Line 20 sets a new valid time by calling member function `setTime`, and line 21 again shows the time in both formats.

Calling `setTime` with Invalid Values

To show that `setTime` validates its arguments, line 25 calls `setTime` with invalid arguments of 99 for the hour, minute and second parameters. We placed this statement in a `try` block (lines 24–26) in case `setTime` throws an `invalid_argument` exception, which it will do in this example. When the exception occurs, it's caught at lines 27–29, and line 28 displays the exception's error message by calling its `what` member function. Line 32 shows the time to confirm that `setTime` did not change the time when invalid arguments were supplied.

9.7.12 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.

An object in memory contains only data, not the class's 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.



Checkpoint

- 1 *(Code)* Write the preprocessor directive you would place in a header to prevent the header's code from being included in the same source-code file more than once.

Answer: `#pragma once`

- 2 *(Fill-in)* The _____ 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.

Answer: interface

- 3 *(True/False)* 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.

Answer: True.

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 correctly creates and manipulates `Time` objects.

The last step to create the executable `Time` application is to link

- the object code for the `main` function (that is, the client code),
- the object code for class `Time`'s member-function implementations and
- 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.

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

Section 1.11 showed how to compile and run C++ applications that contained one source-code (`.cpp`) file. To compile and link multiple source-code files:¹⁸

- In Microsoft Visual Studio, add to your project (as shown in Section 1.11.1) 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 `g++` or `clang++` at the command line, open a shell and change to the directory containing the program's files. 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.



Checkpoint

- I (True/False) Compilers and IDEs typically invoke the linker for you after compiling your code.

Answer: True.

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 Chapter 20.) Within a class's scope, 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

18. This process changes with C++20 modules, as we'll discuss in Chapter 16.



- an object name,
- a reference to an object,
- a pointer to an object or
- a pointer to a specific class member (discussed briefly in Chapter 20).

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 the compiler inserts an implicit handle called the **this** pointer 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:

```
Account account{}; // an Account object
Account& ref{account}; // ref refers to an Account object
Account* ptr{&account}; // ptr points to an Account object
```

you can invoke member function `deposit` using the dot (.) and arrow (->) member selection operators as follows:

```
account.deposit(123.45); // call deposit via account object's name
ref.deposit(123.45); // call deposit via reference to account object
ptr->deposit(123.45); // call deposit via pointer to account object
```

Again, you should use references in preference to pointers whenever possible. We'll continue to show pointers when required and to prepare you to work with them in the legacy code you'll encounter in industry.



Checkpoint

1 (Fill-in) To access an object's members via a pointer to an object, use the _____ followed by the member name.

Answer: arrow member-selection operator (->).

2 (Code) Assume `timePtr` is a pointer to a `Time` object. Write a statement that uses `timePtr` to call the `Time` object's `setTime` member function with the values 6, 45 and 0.

Answer: `timePtr->setTime(6, 45, 0);`

9.10 Access Functions and Utility Functions

Access Functions

Access functions can read or display data but 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.¹⁹

Utility Functions

A **utility function** (also called a **helper function**) is a **private** member function that supports the operation of a class's other member functions and is 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. Many codebases have naming conventions for **private** member functions, such as preceding their names with an underscore (_).



Checkpoint

1 *(Fill-in)* A(n) _____ is an access function that tests whether a condition is true or false.

Answer: predicate function.

2 *(Fill-in)* A(n) _____ typically contains code that would otherwise be duplicated in several other member functions

Answer: utility function.

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.

9.11.1 Class Time

Like other functions, constructors can specify default arguments. Line 11 of Fig. 9.10 declares a `Time` constructor with the default argument value 0 for each parameter. A constructor with default arguments for all its parameters is also a default constructor—that is, it can be invoked with no arguments. There can be only one default constructor per class.²⁰ Any change to a function's default argument values requires the client code to be recompiled (to ensure that the program still functions correctly).



```

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

```

Fig. 9.10 | Time class containing a constructor with default arguments. (Part 1 of 2.)

19. 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`. Such functions also should be preceded with the attribute `[[nodiscard]]` so the compiler confirms that the return value is used in the caller, rather than ignored.

20. You'll see that C++20 concepts (Chapter 15) enable you to overload any function or member function for use with types that match specific requirements.



```

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 };

```

Fig. 9.10 | Time class containing a constructor with default arguments. (Part 2 of 2.)

Class Time's Constructor

In Fig. 9.11, lines 9–11 define the Time constructor, which calls `setTime` to validate and assign values to the data members. Function `setTime` (lines 14–31) ensures that `hour` is in the range 0–23 and `minute` and `second` are each in the range 0–59. If any argument is out Err of range, `setTime` throws an exception—in which case, the `Time` object will not complete construction and will not exist for use in the program. This version of `setTime` uses separate `if` statements to validate the arguments so we can provide precise error messages, indicating which argument is out of range. Functions `setHour`, `setMinute` and `setSecond` (lines 34–40) each call `setTime`. Each passes its argument and the current values of the other two data members. For example, to change `m_hour`, `setHour` passes its `hour` argument and the current values of `m_minute` and `m_second`.

```

1 // Fig. 9.11: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 #include <stdexcept>
5 #include <iomanip>
6 #include "Time.h" // include definition of class Time from Time.h
7
8 // Time constructor initializes each data member
9 Time::Time(int hour, int minute, int second) {
10    setTime(hour, minute, second);
11 }

```

Fig. 9.11 | Member-function definitions for class Time. (Part 1 of 2.)

```
12 // set new Time value using 24-hour time
13 void Time::setTime(int hour, int minute, int second) {
14     // validate hour, minute and second
15     if (hour < 0 || hour >= 24) {
16         throw std::invalid_argument{"hour was out of range"};
17     }
18
19     if (minute < 0 || minute >= 60) {
20         throw std::invalid_argument{"minute was out of range"};
21     }
22
23     if (second < 0 || second >= 60) {
24         throw std::invalid_argument{"second was out of range"};
25     }
26
27     m_hour = hour;
28     m_minute = minute;
29     m_second = second;
30 }
31
32 // set hour value
33 void Time::setHour(int hour) {setTime(hour, m_minute, m_second);}
34
35 // set minute value
36 void Time::setMinute(int minute) {setTime(m_hour, minute, m_second);}
37
38 // set second value
39 void Time::setSecond(int second) {setTime(m_hour, m_minute, second);}
40
41 // return hour value
42 int Time::getHour() const {return m_hour;}
43
44 // return minute value
45 int Time::getMinute() const {return m_minute;}
46
47 // return second value
48 int Time::getSecond() const {return m_second;}
49
50
51 // return Time as a string in 24-hour format (HH:MM:SS)
52 std::string Time::to24HourString() const {
53     return std::format("{:02d}:{:02d}:{:02d}",
54         getHour(), getMinute(), getSecond());
55 }
56
57 // return Time as a string in 12-hour format (HH:MM:SS AM or PM)
58 std::string Time::to12HourString() const {
59     return std::format("{}:{}{:02d}:{:02d} {}",
60         ((getHour() % 12 == 0) ? 12 : getHour() % 12),
61         getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
62 }
```

Fig. 9.11 | Member-function definitions for class Time. (Part 2 of 2.)

 CG The C++ Core Guidelines provide many constructor recommendations. We'll see more in the next two chapters. If a class has a **class invariant**²¹—that is, it requires its data members to have specific values or ranges of values (as in class `Time`)—the class should define a constructor that validates its arguments and, if any are invalid, throws an exception to prevent the object from being created.^{22,23,24}

 Err

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 16),
- one with one argument specified (line 17),
- one with two arguments specified (line 18),
- one with three arguments specified (line 19) and
- one with three invalid arguments specified (line 29).

 Err The program displays each object in 24-hour and 12-hour time formats. The program displays an error message for `Time` object `t5` (line 29) because its 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.

```

1 // fig09_12.cpp
2 // Constructor with default arguments.
3 #include <format>
4 #include <iostream>
5 #include <stdexcept>
6 #include <string>
7 #include "Time.h" // include definition of class Time from Time.h
8
9 // displays a Time in 24-hour and 12-hour formats
10 void displayTime(std::string_view message, const Time& time) {
11     std::cout << std::format("{}\n24-hour time: {}{}\n12-hour time: {}{}\n",
12         message, time.to24HourString(), time.to12HourString());
13 }
14
15 int main() {
16     const Time t1{}; // all arguments defaulted
17     const Time t2{2}; // hour specified; minute & second defaulted
18     const Time t3{21, 34}; // hour & minute specified; second defaulted
19     const Time t4{12, 25, 42}; // hour, minute & second specified

```

Fig. 9.12 | Constructor with default arguments. (Part 1 of 2.)

-
21. “Class invariant.” Wikipedia. Wikimedia Foundation. Accessed April 14, 2023. https://en.wikipedia.org/wiki/Class_invariant.
 22. C++ Core Guidelines, “C.40: Define a Constructor if a Class Has an Invariant.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor>.
 23. C++ Core Guidelines, “C.41: A Constructor Should Create a Fully Initialized Object.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-complete>.
 24. C++ Core Guidelines, “C.42: If a Constructor Cannot Construct a Valid Object, Throw an Exception.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-throw>.

```

20      std::cout << "Constructed with:\n\n";
21      displayTime("t1: all arguments defaulted", t1);
22      displayTime("t2: hour specified; minute and second defaulted", t2);
23      displayTime("t3: hour and minute specified; second defaulted", t3);
24      displayTime("t4: hour, minute and second specified", t4);
25
26
27      // attempt to initialize t5 with invalid values
28      try {
29          const Time t5{27, 74, 99}; // all bad values specified
30      }
31      catch (const std::invalid_argument& e) {
32          std::cerr << std::format("t5 not created: {}\n", e.what());
33      }
34  }

```

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 was out of range

```

Fig. 9.12 | Constructor with default arguments. (Part 2 of 2.)

Software Engineering Notes Regarding Class Time's Set and Get Functions and Its Constructor

Time's *set* and *get* functions are called throughout the class's body. In particular, the constructor (Fig. 9.11, lines 9–11) calls *setTime*, and *to24HourString* and *to12HourString* call *getHour*, *getMinute* and *getSecond* in lines 54 and lines 60–61. We could have accessed the class's *private* data in each case directly.

Our internal time representation uses three *ints*, requiring 12 bytes of memory on systems with four-byte *ints*. Consider changing this to the total seconds since midnight—a single *int* requiring only four bytes of memory. If we made this change, only the bodies of functions directly accessing 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.

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 called DRY—"don't repeat yourself."²⁵ 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.



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 properly initializing them.



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.



Checkpoint

1 (*Discussion*) A constructor can call the class's other member functions. Explain why you must be careful when doing this.

Answer: 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.

2 (*True/False*) Making data members `private` ensures the benefits of data integrity.

Answer: False. Actually, the benefits of data integrity are not automatic simply because data members are `private`. You must provide appropriate validity checking.

3 (*Discussion*) Consider a `Time` class definition that includes both of the following constructor prototypes:

```
Time(int h = 0, int m = 0, int s = 0);
Time();
```

Can this class be used to default construct a `Time` object? If not, explain why.

Answer: This class cannot be used to default construct a `Time` object because both constructors may be called with no arguments. If a call is ambiguous, the C++ compiler will issue an error message.

9.11.2 Overloaded Constructors and Delegating Constructors

Section 5.14 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.

25. "Don't Repeat Yourself." Wikipedia. Wikimedia Foundation. Accessed April 14, 2023, https://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

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 (and appropriate definitions in `Time.cpp`):

```
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
```

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 called 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.²⁶



The first three of the four `Time` preceding constructor declarations 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 in:

```
Time::Time() : Time{0, 0, 0} {}
Time::Time(int hour) : Time{hour, 0, 0} {}
Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```



Checkpoint

- 1** (*True/False*) Overloaded constructors allow objects to be initialized with different types and/or numbers of arguments.

Answer: True.

- 2** (*True/False*) Constructors cannot call other constructors in the same class.

Answer: False. Actually, delegating constructors can call other constructors in the same class to delegate their work.

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`.

A class's destructor is called implicitly when an object is destroyed, typically when program control 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 and cleaning up other resources used by the object) 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.²⁷ In Chapter 12, we'll explain why exceptions should not be thrown from destructors.

26. C++ Core Guidelines, “C.51: Use Delegating Constructors to Represent Common Actions for all Constructors of a Class.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-delegating>.

27. We'll see that such a default destructor also destroys class objects that are created through inheritance (Chapter 10).



Checkpoint

1 (*True/False*) A class's destructor is called implicitly when an object is destroyed, typically when program control leaves the scope in which that object was created.

Answer: True.

2 (*Fill-in*) Every class has one destructor. If you do not explicitly define one, the compiler defines a _____ destructor that invokes any class-type data members' destructors.

Answer: default.

9.13 When Constructors and Destructors Are Called

The order in which constructors and destructors 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.

Constructors and Destructors for Objects in Global Scope



Constructors are called for objects defined in the global scope (also called global namespace scope) before executing any other function in that file. The execution order of global object constructors among multiple files is not guaranteed, so global objects in separate files should not depend on one another. When `main` terminates via a `return` statement or by reaching its closing brace, the corresponding destructors are called in the reverse order of their construction.

Destructors on Program Termination



The `exit` function often is called to terminate a program immediately when a fatal unrecoverable error occurs. In this case, the destructors of local objects do not execute.²⁸ 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 program termination.²⁹

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 `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

28. “`std::exit`.” Accessed April 14, 2023. <https://en.cppreference.com/w/cpp/utility/program/exit>.

29. “`std::abort`.” Accessed April 14, 2023. <https://en.cppreference.com/w/cpp/utility/program/abort>.

the reverse order of their creation. Destructors are not called for `static` objects if the program terminates with a call to `abort`.

Demonstrating When Constructors and Destructors Are Called

The program in 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. Most classes you create will not require custom destructors unless objects of the class manage dynamically allocated memory (discussed in Chapter 11).

Figure 9.13 declares class `CreateAndDestroy`. Lines 13–14 declare the class's data members—an integer (`m_ID`) and a `string` (`m_message`) to identify each object in the program's output.

```
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};
```

Fig. 9.13 | `CreateAndDestroy` class definition.

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 18) 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.

```
1 // Fig. 9.14: CreateAndDestroy.cpp
2 // CreateAndDestroy class member-function definitions.
3 #include <format>
4 #include <iostream>
5 #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
6
7 // constructor sets object's ID number and descriptive message
8 CreateAndDestroy::CreateAndDestroy(int ID, std::string_view message)
9    : m_ID{ID}, m_message{message} {
10    std::cout << std::format("Object {} constructor runs  {}\n",
11                           m_ID, m_message);
12}
```

Fig. 9.14 | `CreateAndDestroy` class member-function definitions. (Part 1 of 2.)

```

13 // destructor
14 CreateAndDestroy::~CreateAndDestroy() {
15     // output newline for certain objects; helps readability
16     std::cout << std::format("{}Object {} destructor runs  {}\n",
17         (m_ID == 1 || m_ID == 6 ? "\n" : ""), m_ID, m_message);
18 }
19 }
```

Fig. 9.14 | CreateAndDestroy class member-function definitions. (Part 2 of 2.)

Figure 9.15 defines the object `first` (line 9) in the 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.

```

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
7 void create(); // prototype
8
9 const CreateAndDestroy first{1, "(global before main)" }; // global object
10
11 int main() {
12     std::cout << "\nMAIN FUNCTION: EXECUTION BEGINS\n";
13     const CreateAndDestroy second{2, "(local in main)" };
14     static const CreateAndDestroy third{3, "(local static in main)" };
15
16     create(); // call function to create objects
17
18     std::cout << "\nMAIN FUNCTION: EXECUTION RESUMES\n";
19     const CreateAndDestroy fourth{4, "(local in main)" };
20     std::cout << "\nMAIN FUNCTION: EXECUTION ENDS\n";
21 }
22
23 // function to create objects
24 void create() {
25     std::cout << "\nCREATE FUNCTION: EXECUTION BEGINS\n";
26     const CreateAndDestroy fifth{5, "(local in create)" };
27     static const CreateAndDestroy sixth{6, "(local static in create)" };
28     const CreateAndDestroy seventh{7, "(local in create)" };
29     std::cout << "\nCREATE FUNCTION: EXECUTION ENDS\n";
30 }
```

```

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)
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 1 of 2.)

```
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)
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 2 of 2.)

Function `main` (lines 11–21) defines three objects. Objects `second` (line 13) and `fourth` (line 19) are local objects, and object `third` (line 14) is a `static` local object. The constructor for each object 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 24–30) defines three objects—`fifth` (line 26) and `seventh` (line 28) are local automatic objects, and `sixth` (line 27) 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 non-`static` objects are destroyed. As an exercise, modify this program to call `create` twice—you'll see that the `static` object `sixth`'s constructor is called once when `create` is called the first time.



Checkpoint

- 1** (*True/False*) Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but global and `static` objects can alter the order in which destructors are called.

Answer: True.

- 2** (*True/False*) Constructors are called for objects defined in the global scope (also called global namespace scope) before executing any other function in that file.

Answer: True.

- 3** (*True/False*) Global and `static` objects are destroyed in the order of their creation.

Answer: False. Actually, global and `static` objects are destroyed in the reverse order of their creation.

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`, as we did for the `getName` member function of our `Account` class earlier in this chapter, the reference is a nonmodifiable *lvalue* and cannot be used to modify the data. However, subtle errors can occur if the reference return type is not declared `const`.

The program of Figs. 9.16–9.18 is a mechanical example that uses a simplified `Time` class to demonstrate the risk of returning a reference to a `private` data member. Member function `badSetHour` (declared in Fig. 9.16 in line 12 and defined in Fig. 9.17 in lines 30–33) returns an `int&` to the `m_hour` data member. 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 like the `private` data member as an *lvalue* in an assignment statement. So, clients of the class can overwrite the class's `private` data at will! A similar problem would occur if the function returned a pointer to the `private` data.

```

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 hour); // dangerous reference return
13 private:
14    int m_hour{0};
15    int m_minute{0};
16    int m_second{0};
17 };

```

Fig. 9.16 | Time class declaration.

```

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

```

Fig. 9.17 | Time class member-function definitions. (Part I of 2.)

```

6 // set new Time value using 24-hour time
7 void Time::setTime(int hour, int minute, int second) {
8     // validate hour, minute and second
9     if (hour < 0 || hour >= 24) {
10         throw std::invalid_argument{"hour was out of range"};
11     }
12
13     if (minute < 0 || minute >= 60) {
14         throw std::invalid_argument{"minute was out of range"};
15     }
16
17     if (second < 0 || second >= 60) {
18         throw std::invalid_argument{"second was out of range"};
19     }
20
21     m_hour = hour;
22     m_minute = minute;
23     m_second = second;
24 }
25
26 // return hour value
27 int Time::getHour() const {return m_hour;}
28
29 // poor practice: returning a reference to a private data member
30 int& Time::badSetHour(int hour) {
31     setTime(hour, m_minute, m_second);
32     return m_hour; // dangerous reference return
33 }
```

Fig. 9.17 | Time class member-function definitions. (Part 2 of 2.)

Figure 9.18 declares Time object *t* (line 9) and reference *hourRef* (line 12), which we initialize with the reference returned by *t*.*badSetHour*(20). Lines 14–15 display *hourRef*'s value to show that *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 the invalid value 30. Lines 17–18 call *getHour* to show that assigning to *hourRef* modified *t*'s private data. Line 22 uses the *badSetHour* function call as an *lvalue* and assigns the invalid value 74 to the reference the function returns. Lines 24–25 call *getHour* again to show that line 22 modified the private data in the *Time* object *t*.

Err

```

1 // fig09_18.cpp
2 // public member function that
3 // returns a reference to private data.
4 #include <iostream>
5 #include <format>
6 #include "Time.h" // include definition of class Time
7
8 int main() {
9     Time t{}; // create Time object
10 }
```

Fig. 9.18 | public member function that returns a reference to private data. (Part 1 of 2.)

```

11 // initialize hourRef with the reference returned by badSetHour
12 int& hourRef{t.badSetHour(20)}; // 20 is a valid hour
13
14 std::cout << std::format(
15     "Valid hour before modification: {}\n", hourRef);
16 hourRef = 30; // use hourRef to set invalid value in Time object t
17 std::cout << std::format(
18     "Invalid hour after modification: {}\n\n", 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 std::cout << "After using t.badSetHour(12) as an lvalue, "
25     << std::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

```

Fig. 9.18 | public member function that returns a reference to private data. (Part 2 of 2.)



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 `MyArray` class in Section 11.6.



Checkpoint

1 (*True/False*) 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.

Answer: True.

2 (*Discussion*) What problem is caused by returning a reference or a pointer to a private data member?

Answer: It breaks the class's encapsulation, making the client code dependent on the class's data representation.

9.15 Default Assignment Operator

The assignment operator (=) can assign an object to another object of the same type. The **default assignment operator**³⁰ generated by the compiler copies each data member of the right operand into the same data member in the left operand. Figures 9.19 and 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.

30. This is actually the default copy assignment operator. In Chapter 11, we'll distinguish between the copy assignment operator and the move assignment operator.

```

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     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.

```

1 // Fig. 9.20: Date.cpp
2 // Date class member-function definitions.
3 #include <format>
4 #include <string>
5 #include "Date.h" // include definition of class Date from Date.h
6
7 // Date constructor (should do range checking)
8 Date::Date(int year, int month, int day)
9     : m_year{year}, m_month{month}, m_day{day} {}
10
11 // return string representation of a Date in the format yyyy-mm-dd
12 std::string Date::toString() const {
13     return std::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
14 }

```

Fig. 9.20 | Date class member-function definitions.

```

1 // fig09_21.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using the default assignment operator.
4 #include <format>
5 #include <iostream>
6 #include "Date.h" // include definition of class Date from Date.h
7 using namespace std;
8
9 int main() {
10     const Date date1{2006, 7, 4};
11     Date date2{2022, 1, 1};
12
13     std::cout << std::format("date1: {}\ndate2: {}\n\n",
14                             date1.toString(), date2.toString());

```

Fig. 9.21 | Class objects can be assigned to each other using the default assignment operator.
(Part I of 2.)

```

15     date2 = date1; // uses the default assignment operator
16     std::cout << std::format("After assignment, date2: {}\n",
17         date2.toString());
18 }

```

date1: 2006-07-04
 date2: 2022-01-01

After assignment, date2: 2006-07-04

Fig. 9.21 | Class objects can be assigned to each other using the default assignment operator.

(Part 2 of 2.)

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.³¹ For example, you can copy date1’s data-member values into a new Date object with the statement:

```
Date date3{date1};
```



Checkpoint

I *(Fill-in)* 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(n) _____ to copy the original object’s data into the new object.

Answer: copy constructor.

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, so you should declare them **const**. Any attempt to modify a **const** object results in a compilation error. The statement

```
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.



C++ disallows calling a member function on a **const** object unless that member function is declared **const**. So, declare as **const** any member function that does not modify the object on which it’s called.

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

31. In Chapter 11, we’ll discuss cases in which the compiler uses move constructors, rather than copy constructors.

reclaimed by the system. So, declaring a constructor or destructor `const` is a compilation error. The “`constness`” of a `const` object is enforced throughout the object’s lifetime after the object is constructed. 

Using `const` and Non-`const` Member Functions

The program of Fig. 9.22 uses a copy of class `Time` from Figs. 9.10 and 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).

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.

```

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}; // constexpr 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 }
```

clang++ compiler error messages:

```

fig09_22.cpp:11:4: error: 'this' argument to member function 'setHour' has
type 'const Time', but function is not marked const
    noon.setHour(12);           // const       non-const
    ^
./Time.h:15:9: note: 'setHour' declared here
    void setHour(int hour); // set hour (after validation)
    ^
```

Fig. 9.22 | `const` objects and `const` member functions. (Part I of 2.)

```
fig09_22.cpp:15:4: error: 'this' argument to member function 'to12HourString' has type 'const Time', but function is not marked const
    noon.to12HourString(); // const      non-const
    ^~~~

./Time.h:25:16: note: 'to12HourString' declared here
    std::string to12HourString(); // 12-hour time format string
    ^
2 errors generated.
```

Fig. 9.22 | `const` objects and `const` member functions. (Part 2 of 2.)



A constructor must be a non-`const` member function, but it can still initialize a `const` object (Fig. 9.22, line 7). Recall from Fig. 9.11 that the `Time` constructor's definition calls non-`const` member function `setTime` to initialize a `Time` object. Invoking a non-`const` member function from the constructor for a `const` object is allowed. Again, 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. This fact is not sufficient—you must explicitly declare the function `const` for this call to be allowed by the compiler.



Checkpoint

1 *(Fill-in)* C++ disallows calling a member function on a `const` object unless that member function is _____.

Answer: `const`.

2 *(True/False)* 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, declaring a constructor or destructor `const` is a compilation error.

Answer: True.

3 *(True/False)* The “constness” of a `const` object is enforced throughout the object's lifetime after the object is constructed.

Answer: True.

9.17 Composition: Objects as Members of Classes



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** and is sometimes referred to as a **has-a relationship**—a class can have objects of other classes as members.³² You've already used composition in this chapter's `Account` class examples. Class `Account` contained a `string` object as a data member.

You've seen how to pass arguments to a constructor. Now, let's see how a class's constructor can pass arguments to member-object constructors via member initializers. The

32. 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.

next program uses classes `Date` (Figs. 9.23 and 9.24) and `Employee` (Figs. 9.25 and 9.26) to demonstrate composition. Class `Employee`'s definition (Fig. 9.25) has private data members `m.firstName`, `m.lastName`, `m.birthDate` and `m.hireDate`. Members `m.birthDate` and `m.hireDate` are objects of class `Date`, which has private data members `m.year`, `m.month` and `m.day`.

```

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     Date(int year, int month, int day);
10    std::string toString() const; // date string in yyyy-mm-dd format
11    ~Date(); // implementation displays 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};

```

Fig. 9.23 | Date class definition.

```

1 // Fig. 9.24: Date.cpp
2 // Date class member-function definitions.
3 #include <array>
4 #include <format>
5 #include <iostream>
6 #include <stdexcept>
7 #include "Date.h" // include Date class definition
8
9 // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date(int year, int month, int day)
12     : m_year{year}, m_month{month}, m_day{day} {
13     if (m_month < 1 || m_month > monthsPerYear) { // validate the month
14         throw std::invalid_argument{"month must be 1-12"};
15     }
16
17     if (!checkDay(day)) { // validate the day
18         throw std::invalid_argument{
19             "Invalid day for current month and year"};
20     }
21

```

Fig. 9.24 | Date class member-function definitions. (Part I of 2.)

```

22     // output Date object to show when its constructor is called
23     std::cout << std::format("Date object constructor: {}\n", toString());
24 }
25
26 // gets string representation of a Date in the form yyyy-mm-dd
27 std::string Date::toString() const {
28     return std::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     std::cout << std::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     // we ignore element 0
40     static const std::array daysPerMonth{
41         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31};
42
43     // determine whether testDay is valid for specified month
44     if (1 <= day && day <= daysPerMonth.at(m_month)) {
45         return true;
46     }
47
48     // February 29 check for leap year
49     if (m_month == 2 && day == 29 && (m_year % 400 == 0 ||
50         (m_year % 4 == 0 && m_year % 100 != 0))) {
51         return true;
52     }
53
54     return false; // invalid day, based on current m_month and m_year
55 }
```

Fig. 9.24 | Date class member-function definitions. (Part 2 of 2.)

```

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
```

Fig. 9.25 | Employee class definition showing composition. (Part 1 of 2.)

```

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  };

```

Fig. 9.25 | Employee class definition showing composition. (Part 2 of 2.)

```

1 // Fig. 9.26: Employee.cpp
2 // Employee class member-function definitions.
3 #include <format>
4 #include <iostream>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // constructor uses member initializer list to pass initializer
9 // values to constructors of member objects
10 Employee::Employee(std::string_view firstName, std::string_view lastName,
11                     const Date &birthDate, const Date &hireDate)
12     : m(firstName{firstName}, m.lastName{lastName},
13           m.birthDate{birthDate}, m.hireDate{hireDate}) {
14     // output Employee object to show when constructor is called
15     std::cout << std::format("Employee object constructor: {} {}\n",
16                             m.firstName, m.lastName);
17 }
18
19 // gets string representation of an Employee object
20 std::string Employee::toString() const {
21     return std::format("{} {} Hired: {} Birthday: {}", m.lastName,
22                         m.firstName, m.hireDate.toString(), m.birthDate.toString());
23 }
24
25 // output Employee object to show when its destructor is called
26 Employee::~Employee() {
27     cout << std::format("Employee object destructor: {}, {}\n",
28                         m.lastName, m.firstName);
29 }

```

Fig. 9.26 | Employee class member-function definitions.

Employee Constructor's Member-Initializer List

The Employee constructor's prototype (Fig. 9.25, lines 11–12) specifies that the constructor has four parameters (`firstName`, `lastName`, `birthDate` and `hireDate`). In the constructor's definition (Fig. 9.26, lines 12–13), 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`. The 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. Again, for clarity, the C++ Core Guidelines recommend listing the member initializers in the order they're declared in the class.³³



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 brace initialize the m_birthDate and m_hireDate objects by passing Date objects to their constructors? As 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 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 9–10), then passes them as arguments to the constructor of the Employee object created in line 11. When an Employee is created, its constructor makes

- two calls to the string class's constructor (line 12 of Fig. 9.26) and
- two calls to the Date class's default copy constructor (line 13 of Fig. 9.26).

Line 13 of Fig. 9.27 displays the Employee's data to show that it was initialized correctly.

```

1 // fig09_27.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <format>
4 #include <iostream>
5 #include "Date.h" // Date class definition
6 #include "Employee.h" // Employee class definition
7
8 int main() {
9     const Date birth{1987, 7, 24};
10    const Date hire{2018, 3, 12};
11    const Employee manager{"Aisha", "Khan", birth, hire};
12
13    std::cout << std::format("\n{}\\n\\n", manager.toString());
14 }
```

```

Date object constructor: 1987-07-24
Date object constructor: 2018-03-12
Employee object constructor: Aisha Khan

Khan, Aisha Hired: 2018-03-12 Birthday: 1987-07-24

Employee object destructor: Khan, Aisha
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.

33. C++ Core Guidelines, “C.47: Define and Initialize Member Variables in the Order of Member Declaration.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order>.

When each `Date` object is created in lines 9–10, the `Date` constructor (lines 11–24 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 11 of Fig. 9.27 causes two `Date` copy-constructor calls (line 13 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 26–29 of Fig. 9.26, respectively) that prints a message when an object of its class is destructed. The destructors help us show that objects are destructed from the outside in. 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 10) and `birth` (line 9), respectively. The outputs confirm that the three objects created in `main` are destructed in the reverse order of their construction. 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 `firstName` and `lastName` objects being destructed.

What Happens When You Do Not Use the Member-Initializer List?

If you do not initialize a member object explicitly, its default constructor will be called implicitly to initialize it. If there is no default constructor, a compilation error occurs. Values set by the default constructor can be changed later—for example, by *set* functions. However, this approach may require significant additional work and time for complex initialization. Initializing member objects via member initializers eliminates the overhead of “doubly initializing” member objects.



Checkpoint

- 1 *(True/False)* A class’s constructor can pass arguments to member-object constructors via member initializers.

Answer: True.

- 2 *(True/False)* Data members are constructed in the order they appear in the member-initializer list.

Answer: False. Actually, the order of the member initializers does not matter. Data members are constructed in the order that they’re declared in the class, 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.

- 3 *(True/False)* Objects are generally destructed from the outside in.

Answer: True.

9.18 friend Functions and friend Classes

A **friend function** has access to a class's **public** and non-public members. A class may have as **friends**

- stand-alone 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, Copy/Move Semantics and Smart Pointers, 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 an existing class **ClassTwo** as friends of class **ClassOne**, place in **ClassOne**'s definition a declaration of the form:

```
friend class ClassTwo;
```

Friendship Rules



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 **friend** 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 the free function **modifyX** as a **friend** of class **Count** (line 8), so **modifyX** can set **Count**'s **private** data member **m_x**.

```

1 // fig09_28.cpp
2 // Friends can access private members of a class.
3 #include <format>
4 #include <iostream>
5

```

Fig. 9.28 | Friends can access **private** members of a class. (Part I of 2.)

```

6 // Count class definition
7 class Count {
8     friend void modifyX(Count& c, int value); // friend declaration
9 public:
10    int getX() const {return m_x;}
11 private:
12    int m_x{0};
13 };
14
15 // function modifyX can modify private data of Count
16 // because modifyX is declared as a friend of Count (line 8)
17 void modifyX(Count& c, int value) {
18     c.m_x = value; // allowed because modifyX is a friend of Count
19 }
20
21 int main() {
22     Count counter{}; // create Count object
23
24     std::cout << std::format("Initial counter.m_x: {}\n", counter.getX());
25     modifyX(counter, 8); // change x's value using a friend function
26     std::cout << std::format("counter.m_x after modifyX: {}\n",
27                             counter.getX());
28 }
```

Initial counter.m_x: 0
 counter.m_x after modifyX: 8

Fig. 9.28 | Friends can access private members of a class. (Part 2 of 2.)

Function `modifyX` (lines 17–19) is a stand-alone (free) function, not a `Count` member function. So, when we call `modifyX` in `main` to modify the `Count` object `counter` (line 25), we must pass `counter` as an argument to `modifyX`. Function `modifyX` can access class `Count`'s private data member `m_x` (line 18) only because the function was declared as a friend of class `Count` (line 8). If you remove the `friend` declaration, you'll receive error messages indicating that function `modifyX` cannot access class `Count`'s private data member `m_x`.



Checkpoint

1 (*Discussion*) Describe what a class may declare as friends.

Answer: A class may declare as friends stand-alone functions, entire classes (and thus all their functions) or specific member functions of other classes.

2 (*True/False*) Member access notions of `public`, `protected` and `private` apply to friend declarations, just as they do to data members and member functions.

Answer: False. Actually, member access notions of `public`, `protected` and `private` do not apply to friend declarations, so friend declarations can be placed anywhere in a class definition.

3 (*Code*) Write a statement that you'd place in `ClassA` to declare all of `ClassB`'s member functions as friends of class `ClassA`.

Answer: `friend class ClassB;`

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 initialized with an implicit argument passed to each of the object's non-static³⁴ member functions.

Using the `this` Pointer to Avoid Naming Collisions

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 a 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, if class `Account` had a `string` data member `name`, we could implement its `setName` function as follows:

```
void Account::setName(std::string_view name) {
    this->name = name; // use this-> to access data member
}
```

where `this->name` represents a data member called `name`. You can avoid such naming collisions by naming your data members differently—for example, using 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`.

³⁴ Section 9.20 introduces `static` class members and explains why the `this` pointer is not implicitly passed to `static` member functions.

```

1 // fig09_29.cpp
2 // Using the this pointer to refer to object members.
3 #include <format>
4 #include <iostream>
5
6 class Test {
7 public:
8     explicit Test(int value);
9     void print() const;
10 private:
11     int m_x{0};
12 };
13
14 // constructor
15 Test::Test(int value) : m_x{value} {} // initialize m_x to value
16
17 // print m_x using implicit then explicit this pointers;
18 // the parentheses around *this are required due to precedence
19 void Test::print() const {
20     // implicitly use the this pointer to access the member m_x
21     std::cout << std::format("      m_x = {}\n", m_x);
22
23     // explicitly use the this pointer and the arrow operator
24     // to access the member m_x
25     std::cout << std::format("  this->m_x = {}\n", this->m_x);
26
27     // explicitly use the dereferenced this pointer and
28     // the dot operator to access the member m_x
29     std::cout << std::format("(*this).m_x = {}\n", (*this).m_x);
30 }
31
32 int main() {
33     const Test testObject{12}; // instantiate and initialize testObject
34     testObject.print();
35 }
```

```

      m_x = 12
this->m_x = 12
(*this).m_x = 12
```

Fig. 9.29 | Using the `this` pointer to refer to object members.

For illustration purposes, member function `print` (lines 19–30) first displays `m_x` using the `this` pointer implicitly (line 21)—only the data member’s name is specified. Then `print` uses two different notations to access `m_x` through the `this` pointer:

- `this->m_x` (line 25) and
- `(*this).m_x` (line 29).

The parentheses around `*this` (line 29) are required because the dot operator (`.`) has higher precedence than the `*` pointer-dereferencing operator. Without the parentheses, the expression `*this.m_x` would be evaluated as `*(this.m_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



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 `setTime`’s body returns a reference to `*this` (line 30). Functions `setHour`, `setMinute` and `setSecond` each call `setTime` and return its `Time&` result.

```

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 <iostream>
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};

```

Fig. 9.30 | Time class modified to enable cascaded member-function calls.

```

1 // Fig. 9.31: Time.cpp
2 // Time class member-function definitions.
3 #include <iostream>
4 #include <stdexcept>
5 #include "Time.h" // Time class definition
6
7 // Time constructor initializes each data member
8 Time::Time(int hour, int minute, int second) {
9     setTime(hour, minute, second);
10 }
11

```

Fig. 9.31 | Time class member-function definitions modified to enable cascaded member-function calls. (Part I of 3.)

```

12 // set new Time value using 24-hour time
13 Time& Time::setTime(int hour, int minute, int second) {
14     // validate hour, minute and second
15     if (hour < 0 || hour >= 24) {
16         throw std::invalid_argument{"hour was out of range"};
17     }
18     if (minute < 0 || minute >= 60) {
19         throw std::invalid_argument{"minute was out of range"};
20     }
21     if (second < 0 || second >= 60) {
22         throw std::invalid_argument{"second was out of range"};
23     }
24     m_hour = hour;
25     m_minute = minute;
26     m_second = second;
27     return *this; // enables cascading
28 }
29
30 // set hour value
31 Time& Time::setHour(int hour) {
32     return setTime(hour, m_minute, m_second);
33 }
34
35 // set minute value
36 Time& Time::setMinute(int minute) {
37     return setTime(m_hour, minute, m_second);
38 }
39
40 // set second value
41 Time& Time::setSecond(int second) {
42     return setTime(m_hour, m_minute, second);
43 }
44
45 // get hour value
46 int Time::getHour() const {return m_hour;}
47
48 // get minute value
49 int Time::getMinute() const {return m_minute;}
50
51 // get second value
52 int Time::getSecond() const {return m_second;}
53
54
55 // return Time as a string in 24-hour format (HH:MM:SS)
56 std::string Time::to24HourString() const {
57     return std::format("{:02d}:{:02d}:{:02d}",
58         getHour(), getMinute(), getSecond());
59 }
60
61 }
62

```

Fig. 9.31 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 3.)

```

63 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
64 std::string Time::to12HourString() const {
65     return std::format("{}:{}{:02d}:{}{:02d} {}",
66         ((getHour() % 12 == 0) ? 12 : getHour() % 12),
67         getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
68 }

```

Fig. 9.31 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 3.)

In Fig. 9.32, we create `Time` object `t` (line 9), then use it in cascaded member-function calls (lines 11 and 19).

```

1 // fig09_32.cpp
2 // Cascading member-function calls with the this pointer.
3 #include <format>
4 #include <iostream>
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    std::cout << std::format("24-hour time: {}\n12-hour time: {}{}\n",
15        t.to24HourString(), t.to12HourString());
16
17    // cascaded function calls
18    std::cout << std::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

```

Fig. 9.32 | Cascading member-function calls with the `this` pointer.

Why does the technique of returning `*this` as a reference work? The dot operator (`.`) groups left-to-right, so line 11

```
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

```
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. We cannot chain another `Time` member-function call after `to12HourString`, because it does not return a `Time&`. However, we could chain a call to a `string` member function because `to12HourString` returns a `string`. Chapter 11 presents practical examples of cascaded function calls, such as in `cout` statements with the `<<` operator and in `cin` statements with the `>>` operator.



Checkpoint

- 1 (*Code*) Assume class `Time` contains a data member `m_totalSeconds`. Show two expressions that could be used to access `m_totalSeconds` via the `this` pointer inside a member function.

Answer: `this->m_totalSeconds`
`(*this).m_totalSeconds`

- 2 (*True/False*) Another use of the `this` pointer is to enable cascaded member-function calls—that is, invoking multiple functions sequentially in the same statement.

Answer: True.

- 3 (*Discussion*) Explain why the technique of returning `*this` as a reference works by explaining how the following line of code works:

```
t.setHour(11).to12HourString();
```

Answer: The dot operator (.) groups left-to-right, so the preceding statement first calls `setHour` to set `t`'s hour to 11. The expression `t.setHour(11)` returns a reference to the updated object `t`, which the statement then uses to call `to12HourString` to get a string representation of the updated object `t`.

9.20 static Class Members: Classwide Data and Member Functions

There is an exception to the rule that each object has its own copy of its class's data members. In some 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 all objects of a class can share a single copy of the data.

Motivating Classwide Data

Let's further explore the need for `static` classwide data with an example. Suppose that we have a video game with `Martians` and other `SpaceCreatures`. Each `Martian` tends to be brave and willing to attack other `SpaceCreatures` 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 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, but only one copy of the `static 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

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. 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 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.

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.

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`, which is initialized to 0 (Fig. 9.33, line 22), 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:

```

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

```

Fig. 9.33 | Employee class definition with a `static` data member to track the number of `Employee` objects in memory. (Part 1 of 2.)

```

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 };

```

Fig. 9.33 | Employee class definition with a static data member to track the number of Employee objects in memory. (Part 2 of 2.)

In Fig. 9.34, line 10 defines static member function `getCount`—note this 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, as shown in the constructor (line 16) and the destructor (line 25).

```

1 // Fig. 9.34: Employee.cpp
2 // Employee class member-function definitions.
3 #include <iostream>
4 #include <iomanip>
5 #include "Employee.h" // Employee class definition
6 using namespace std;
7
8 // define static member function that returns number of
9 // Employee objects instantiated (declared static in Employee.h)
10 int Employee::getCount() {return m_count;}
11
12 // constructor initializes non-static data members and
13 // increments static data member count
14 Employee::Employee(string_view firstName, string_view lastName)
15   : m(firstName(firstName), m(lastName(lastName)) {
16     ++m_count; // increment static count of employees
17     std::cout << std::format("Employee constructor called for {} {}\n",
18       m(firstName, m(lastName));
19 }
20

```

Fig. 9.34 | Employee class member-function definitions. (Part 1 of 2.)

```

21 // destructor decrements the count
22 Employee::~Employee() {
23     std::cout << std::format("~Employee() called for {} {}\n",
24         m(firstName, m.lastName);
25     --m_count; // decrement static count of employees
26 }
27
28 // return first name of employee
29 const string& Employee::getFirstName() const {return m.firstName;}
30
31 // return last name of employee
32 const string& Employee::getLastName() const {return m.lastName;}

```

Fig. 9.34 | Employee class member-function definitions. (Part 2 of 2.)

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 11),
- after two `Employee` objects have been created (line 23) and
- after those `Employee` objects have been destroyed (line 34).

When `Employee` objects are in scope, you also can call `getCount` on those objects. For example, in line 23, we could have written either `e1.getCount()` or `e2.getCount()`. Either would return the current value of class `Employee`'s static `m_count`.

```

1 // fig09_35.cpp
2 // static data member tracking the number of objects of a class.
3 #include <iostream>
4 #include <iomanip>
5 #include "Employee.h" // Employee class definition
6
7 int main() {
8     // no objects exist; use class name and scope resolution
9     // operator to access static member function getCount
10    std::cout << std::format("Initial employee count: {}\n",
11        Employee::getCount()); // use class name
12
13    // the following scope creates and destroys
14    // Employee objects before main terminates
15    {
16        const Employee e1{"Alma", "Rusu"};
17        const Employee e2{"Grilo", "Perez"};
18
19        // two objects exist; call static member function getCount again
20        // using the class name and the scope resolution operator
21        std::cout << std::format(
22            "Employee count after creating objects: {}\n\n",
23            Employee::getCount());

```

Fig. 9.35 | static data member tracking the number of objects of a class. (Part 1 of 2.)

```
24
25     std::cout << std::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 std::cout << std::format(
33     "Employee count after objects are destroyed: {}\n",
34     Employee::getCount());
35 }
```

```
Initial employee count: 0
Employee constructor called for Alma Rusu
Employee constructor called for Grilo Perez
Employee count after creating objects: 2

Employee 1: Alma Rusu
Employee 2: Grilo Perez

~Employee() called for Grilo Perez
~Employee() called for Alma Rusu
Employee count after objects are destroyed: 0
```

Fig. 9.35 | static data member tracking the number of objects of a class. (Part 2 of 2.)

Lines 15–28 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 16–17). As each constructor executes, it increments class `Employee`’s static data member `m_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 `m_count`.

static Member Function Notes

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.



Info

A `static` member function cannot be declared `const`. The `const` qualifier applies to the object the `this` pointer points to and indicates that a function cannot modify the contents of that object. However, `static` member functions exist and operate independently of any objects of the class, so declaring a `static` member function `const` is a compilation error.



Err



Checkpoint

1 (*Fill-in*) 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 _____ to the member name.

Answer: scope resolution operator (:))

2 (*Discussion*) Explain how to access a **private static** or **protected static** data member when no objects of the class exist,

Answer: Provide a **public static** member function and call the function by prefixing its name with the class name and scope resolution operator.

3 (*Discussion*) Explain why a **static** member function does not have a **this** pointer.

Answer: This is 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.

9.21 Aggregates in C++20

Section 9.4.1 of the C++ standard document

<http://wg21.link/n4861>

describes an **aggregate type** as 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 **private** (Chapter 10), **protected** (Chapter 10) or **virtual** (Chapter 10) base classes.

You can define an aggregate using a class in which all the data is **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:

```
struct Record {
    int account;
    string first;
    string last;
    double balance;
};
```



The C++ Core Guidelines recommend using **class** rather than **struct** if any data member or member function needs to be non-**public**.³⁵

³⁵. C++ Core Guidelines, “C.8: Use **class** Rather Than **struct** if Any Member Is Non-Public.” Accessed April 14, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-class>.



Checkpoint

- 1 *(Code)* Define an aggregate type named `Point` containing `public int` data members named `x` and `y`.

Answer:

```
struct Point {
    int x;
    int y;
};
```

- 2 *(True/False)* The C++ Core Guidelines recommend using `struct` rather than `class` if any data member or member function needs to be non-public.

Answer: False. Actually, the C++ Core Guidelines recommend using `class` rather than `struct` if any data member or member function needs to be non-public.

9.21.1 Initializing an Aggregate

You can initialize an object of aggregate type `Record` (above) as follows:

```
Record record{100, "Abilio", "Saiu", 123.45};
```

If you initialize an aggregate-type object with fewer initializers than there are data members in the object, as in

```
Record record{0, "Abilio", "Saiu"};
```

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.



Checkpoint

- 1 *(Code)* Use the aggregate type `Point` you created previously to create a `Point` object with its `x` and `y` data members initialized to `10`.

Answer: `Point point{10, 10};`

- 2 *(Discussion)* Explain what happens if you initialize an aggregate-type object with fewer initializers than there are data members in the object.

Answer: The remaining data members are initialized with in-class initializers if they have them. Otherwise, they are initialized with empty braces (`{}`), setting fundamental-type variables to `0` and `bools` to `false`, and value initializing objects.

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 `struct Record` definition from the beginning of Section 9.21, we could initialize a `Record` object as follows:

```
Record record{.first{"Couro"}, .last{"Malonga"}};
```

which explicitly initializes only a subset of the data members. Each explicitly named data member is preceded by a dot (.). The identifiers 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 "Couro" and "Malonga", 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 Initializers³⁶

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.



Checkpoint

- I *(Code)* Use a designated initializer to create a `Point` object with the `x`-coordinate 10 and the `y`-coordinate set to its default value.

Answer: `Point point{.x{10}};`

9.22 Concluding Our Objects Natural Case Study Track: Studying the Vigenère Secret-Key Cipher Implementation

In the preceding chapters, we presented nine Objects Natural case studies in which you were able to perform powerful tasks without knowing how to build custom classes. In Section 5.19's Objects Natural case study, we



- introduced cryptography for securing communications, and
- discussed the Caesar substitution cipher and Vigenère secret-key cipher.^{37,38}

Here, we'll present the Vigenère secret-key cipher algorithm and study our `Cipher` class that implements the algorithm. You'll see that the class contains 171 lines of code, including blank lines and comments. This class took significant effort for us to design, write, test and debug. Yet, you were able to use this implementation in Section 5.19 to conveniently explore a nice example of private-key cryptography.

So you've learned that an important part of becoming a C++ developer is to “avoid reinventing the wheel”—take advantage of the enormous numbers of class libraries that the programming community has developed over C++'s four decades. A huge number of these are free and open source and available to you on sites like GitHub. Each repository contains a library's, project's or application's source files, and C++ library repositories often contain many classes.

36. “Designated Initialization.” Accessed April 14, 2023. <http://wg21.link/p0329r0>.

37. “Crypto Corner—Vigenère Cipher.” Accessed April 14, 2023. <https://crypto.interactive-maths.com/vigenegraphere-cipher.html>.

38. “Vigenère cipher.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher.

You've also worked with various existing classes from the C++ Standard Library, such as `string`, `array`, `vector`, `span` and `string_view`, most of which were included in our Objects Natural case studies. In Chapters 13–20, we'll tour other key aspects of the C++ Standard Library, including:

- Chapter 11 and Chapter 20—**Smart pointers** which provide additional functionality beyond that of built-in pointers to help you avoid errors.
- Chapter 12—Exceptions.
- Chapter 13—Standard Library **containers** and **iterators**.
- Chapter 14—Standard Library **algorithms** and **C++20 ranges and views**.
- Chapter 15—**C++20 Concepts** and various class and function templates used in **compile-time programming** with templates.
- Chapters 17 and 18—**C++ prepackaged parallel algorithms**, and **concurrency** and **C++20 Coroutines** capabilities for building programs that can run faster (often much faster) on today's **multi-core computer architectures**. We also demonstrate some of C++'s `<chrono>` library features to profile the performance improvement.
- Chapter 19—**Input/output streams** and **C++20 text formatting**.
- Chapter 20—More **smart pointers**, additional **compile-time programming** capabilities and more.

Using Our Cipher Class

Figure 9.36 contains a modified version of the `main` application presented in Fig. 5.18 that demonstrates class `Cipher` with the plaintext "Welcome to encryption". We'll discuss encrypting and decrypting this text.

```
1 // fig09_36.cpp
2 // Encrypting and decrypting text with a Vigenère cipher.
3 #include "cipher.h"
4 #include <iostream>
5 #include <string>
6
7 int main() {
8     std::string plainText;
9     std::cout << "Enter the text to encrypt:\n";
10    std::getline(std::cin, plainText);
11
12    std::string secretKey;
13    std::cout << "\nEnter the secret key:\n";
14    std::getline(std::cin, secretKey);
```

Fig. 9.36 | Encrypting and decrypting text with a Vigenère cipher. (Part 1 of 2.)

```

15
16     Cipher cipher;
17
18     // encrypt plainText using secretKey
19     std::string cipherText{cipher.encrypt(plainText, secretKey)};
20     std::cout << "\nEncrypted:\n" << cipherText << '\n';
21
22     // decrypt cipherText
23     std::cout << "\nDecrypted:\n" " "
24         << cipher.decrypt(cipherText, secretKey) << '\n';
25 }
```

Enter the text to encrypt:
Welcome to encryption

Enter the secret key:
XMWUJBVYHXZ

Encrypted:
Tqhwxnz rv bmzdujcjl

Decrypted:
Welcome to encryption

Fig. 9.36 | Encrypting and decrypting text with a Vigenère cipher. (Part 2 of 2.)

Class Cipher

Figure 9.37 defines the `Cipher` class, which we separated into parts for discussion purposes. The first part (lines 1–14) shows the headers our Vigenère implementation requires and the beginning of the class definition.

```

1 // Fig. 9.37: cipher.h
2 // Vigenère cipher implementation.
3 #pragma once
4 #include <algorithm>
5 #include <array>
6 #include <iostream>
7 #include <string>
8 #include <string_view>
9 #include <cctype>
10 #include <stdexcept>
11 #include <gs1/gs1>
12
13 class Cipher {
14 public:
```

Fig. 9.37 | Vigenère cipher implementation.

Vigenère Square

As we mentioned in Section 5.19, simple substitution ciphers like the Caesar cipher are relatively easy to decrypt. For example, “e” is the most frequently used English letter.³⁹ So, you could study English ciphertext and assume that the most frequently appearing character probably is an “e.”

The Vigenère secret-key cipher uses letters from the plaintext and a secret key to locate replacement characters in **26 separate Caesar ciphers**—one for each letter of the alphabet. These 26 ciphers form a 26-by-26 two-dimensional array called the **Vigenère square**:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

You look up substitutions using the bold letters at the left and top that label the rows and columns. Our `Cipher` class's `private` data, defined at the bottom of the class definition, consists of

- a static constant `m_size` (declared at line 153), representing the number of rows and columns in this Vigenère square and
- a two-dimensional `std::array` `m_square` (declared at line 156), representing the Vigenère square.

39. “What is the most common letter?” Accessed April 14, 2023. <https://www.thesaurus.com/e/ways-to-say/most-common-letter/>.

Cipher Constructor

The Cipher constructor (lines 16–39) initializes `m_square` with the 26 Caesar ciphers.

```

15    // constructor to initialize the Vigenère square
16    Cipher() {
17        // array to store the 26 characters A-Z that will be
18        // used to initialize each row of the Vigenère square
19        std::array<char, m_size> alphabet{};
20
21        // fill alphabet with A-Z
22        for (size_t i{0}; i < m_size; ++i) {
23            // convert 'A' + i to a char and place in alphabet
24            alphabet.at(i) = gsl::narrow_cast<char>('A' + i);
25        }
26
27        // copy alphabet into row 0 of the Vigenère square
28        m_square.at(0) = alphabet;
29
30        // for each remaining row of the Vigenère square, move alphabet's
31        // first letter to the end then copy alphabet into the row
32        for (int row{1}; row < m_size; ++row) {
33            // rotate alphabet, moving its first letter to the end
34            std::ranges::rotate(alphabet, std::begin(alphabet) + 1);
35
36            // copy alphabet into current row of the Vigenère square
37            m_square.at(row) = alphabet;
38        }
39    }
40

```

The constructor operates as follows:

- Line 19 defines the `char` array `alphabet` that will initially store the letters A – Z.
- Lines 22–25 fill `alphabet` with A – Z. Recall from Chapter 8 that 'A' has the value 65 in the underlying character set. Adding the values 0 – 25 to 'A' produces the integer values of 'A' through 'Z'. Line 24 uses `gsl::narrow_cast` (Section 5.6) to convert the `int` value to a `char` and place it in `alphabet`.
- Line 28 copies `alphabet`'s elements into row 0 of `m_square`.
- For each of the Vigenère square's remaining rows, lines 32–38 move `alphabet`'s first letter to the end then copy `alphabet` into the current row.
- Line 34 uses the C++20 ranges algorithm `rotate` to move `alphabet`'s first letter to the end and the rest of the its elements toward the beginning, thus “rotating” the elements. The first argument (`alphabet`) is the range to rotate and the second indicates the first of the range's remaining elements that should rotate toward the beginning. The expression `std::begin(alphabet) + 1` represents the element one position after the beginning of `alphabet`—that is, its second element. Elements positioned before the second argument move to the end of the range.

So, if `alphabet` initially contains A–Z,

- after the first `rotate` call, it will contain B–Z followed by A,
- after the second call, it will contain C–Z followed by A–B,
- after the second call, it will contain D–Z followed by A–C, etc.

Recall from Section 8.19 that a benefit of the `std::ranges` algorithms is that you do not need to specify the beginning and end of the range to iterate over—the `std::ranges` algorithms handle this for you, simplifying your code.

Secret-Key Requirements

For the Vigenère cipher described here, the secret key must contain only letters. Like passwords, the secret key should not be easy to guess. To create the ciphertext shown in Fig. 9.36, we used as our secret key the following 11 randomly selected characters:

```
XMWUJBVYHXZ
```

Your key can have as many characters as you like. The person decrypting the ciphertext must know the secret key used to create the ciphertext.⁴⁰ Presumably, you'd provide that in advance—possibly in a face-to-face meeting. The secret key must be carefully guarded.

The Vigenère Cipher Encryption Algorithm

To see how the Vigenère cipher works, let's use the key "XMWUJBVYHXZ" and encrypt the plaintext string:

```
Welcome to encryption
```

Our encryption and decryption implementations preserve the plaintext's original case. We chose to pass non-letters in the plaintext—like spaces, digits and punctuation—through to the ciphertext and vice versa.

First, we repeat the secret key until the length matches the plaintext:

<i>Plaintext:</i>	W e l c o m e t o e n c r y p t i o n
<i>Repeating key text:</i>	X M W U J B V Y H X Z X M W U J B V Y

In this diagram, we highlighted in light blue the secret key, then in darker blue the secret key's eight repeated letters.

We begin the encryption by using the first letter in the repeating key text ('X') to select a row in the Vigenère square and using the first letter in the plaintext ('W') to select a column. The intersection of that row and column (highlighted below) contains the letter to substitute in the ciphertext for 'W'—in this case, 'T':

40. There are many websites offering Vigenère cipher decoders that attempt to decrypt ciphertext without the original secret key. We tried several, but none restored our original text.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

This process continues for each pair of letters from the secret key and the plaintext:

Plaintext:	W	e	l	c	o	m	e		t	o		e	n	c	r	y	p	t	i	o	n				
Repeating key text:	X	M	W	U	J	B	V		Y	H		X	Z	X	M	W	U	J	B	V	Y				
Ciphertext:	T	q	h	w	x	n	z		r	v		b	m	z	d	u	j	c	j	j	l				

Member Function encrypt

Function `encrypt` (lines 43–92) receives the plaintext to encrypt and the secret key and returns a string containing the encrypted text:

```

41 // encrypt receives the plaintext and secret key, applies
42 // the Vigenère cipher returns the encrypted ciphertext
43 std::string encrypt(
44     std::string_view plaintext, std::string_view secret) {
45
46     checkKey(secret); // ensure secret key contains only letters
47
48     std::string ciphertext{}; // stores encrypted text
49     size_t keyIndex{0}; // current letter index in secret key
50
51     for (size_t i = 0; i < plaintext.size(); ++i) {
52         char c = plaintext[i];
53         if (c == ' ') { // ignore spaces
54             ciphertext += ' ';
55             continue;
56         }
57
58         size_t offset = keyIndex % secret.size();
59         char keyChar = secret[offset];
60
61         if (isupper(c)) { // uppercase letter
62             ciphertext += ((c - 'A' + keyChar) % 26) + 'A';
63         } else { // lowercase letter
64             ciphertext += ((c - 'a' + keyChar) % 26) + 'a';
65         }
66
67         keyIndex++;
68     }
69
70     return ciphertext;
71 }
```

(continued...)

```

51     // iterate through each character in plaintext
52     for (size_t i{0}; i < plaintext.size(); ++i) {
53         // determine whether character at i is lowercase so we can
54         // place a corresponding lowercase letter in the ciphertext
55         const bool lower{std::islower(plaintext.at(i)) ? true : false};
56
57         // convert currentChar to uppercase; for uppercase
58         // letters and non-letters the character remains the same
59         const char currentChar{gsl::narrow_cast<const char>(
60             std::toupper(plaintext.at(i)))};
61
62         // if the current character is a letter, encrypt it
63         // and add it to cipherText in its original case
64         if ('A' <= currentChar && currentChar <= 'Z') {
65             // to get the row index in the Vigenère square, select the
66             // character at keyIndex in the secret key, convert it to
67             // uppercase and subtract 'A' from it
68             const int row{std::toupper(secret.at(keyIndex)) - 'A'};
69
70             // increment the keyIndex, ensuring that it is
71             // reset to 0 if keyIndex reaches secret.size()
72             keyIndex = (keyIndex + 1) % secret.size();
73
74             // to get the column index in the Vigenère square,
75             // subtract 'A' from the currentChar
76             const int column{currentChar - 'A'};
77
78             // select the substitute character from the Vigenère square
79             const char substituteChar{m_square.at(row).at(column)};
80
81             // add substituteChar to the ciphertext, ensuring that
82             // it's lowercase if the plaintext character was lowercase
83             ciphertext +=
84                 (lower ? std::tolower(substituteChar) : substituteChar);
85             }
86             else {
87                 ciphertext += currentChar; // add non-letter to ciphertext
88             }
89         }
90
91         return ciphertext; // return the encrypted text
92     }
93

```

The function operates as follows:

- Line 46 calls utility function `checkKey` (defined lines 160–170) to ensure that the secret key contains only letters.
- Line 48 defines the string `ciphertext`, which will store the encrypted text.
- Line 49 defines the `size_t` `keyIndex`, which will keep track of the index of the current character in the secret key. That character is used to select a Vigenère square row.

- For each character in `plaintext`, lines 52–89 either encrypt the letter and add it to `ciphertext` or, if the character is not a letter, simply pass it through to the `ciphertext`.
- Line 55 determines whether the original character was lowercase, so we can maintain that case as we create the ciphertext.
- Lines 59–60 convert the current character to uppercase. If it is already uppercase or is not a letter, `std::toupper` returns the original character. Function `std::toupper`'s result is the `int` representation of the character, so we convert it to a `const char` to place it in `currentChar`.
- If `currentChar` is a letter (line 64), then lines 68–84 encrypt it.
- Line 68 converts the `secret`'s letter at `keyIndex` to uppercase, then subtracts 'A' to produce a value from 0–26 representing the Vigenère square row to select.
- Line 72 increments the `keyIndex` to the next position in `secret`, setting it back to 0 if we've reached the end of `secret`.
- Line 76 calculates the `column` index by subtracting 'A' from `currentChar` to produce a value from 0–26 representing the Vigenère square column to select.
- Line 79 performs the Vigenère cipher encryption algorithm for `currentChar` and stores its ciphertext equivalent in `substituteChar`.
- Lines 83–84 then append `substituteChar` to `ciphertext` using `currentChar`'s original case.
- If `currentChar` was not a letter, line 87 simply appends it to `ciphertext`.

Decrypting with the Vigenère Cipher

The decryption process returns the ciphertext to the original plaintext. It's similar to what we described above and **requires the same secret key used to encrypt the text**. Like the encryption algorithm, the decryption algorithm cycles through the secret key's letters. So, again, we repeat the secret key until the length matches the ciphertext:

<i>Ciphertext:</i>	T q h w x n z r v b m z d u j c j j l
<i>Repeating key text:</i>	X M W U J B V Y H X Z X M W U J B V Y

We begin the decryption by using the first letter in the repeating key text ('X') to select a row in the Vigenère square. Next, we locate within that row the first letter in the ciphertext ('T'). Finally, we replace the ciphertext letter with the plaintext letter at the top of that column ('W'), as highlighted in the Vigenère square below:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	Y	

This process continues for each pair of letters from the secret key and the ciphertext:

Ciphertext:	T	q	h	w	x	n	z	r	v	b	m	z	d	u	j	c	j	j	l
Repeating key text:	X	M	W	U	J	B	V	Y	H	X	Z	X	M	W	U	J	B	V	Y
Plaintext:	W	e	l	c	o	m	e	t	o	e	n	c	r	y	p	t	i	o	n

Member Function decrypt

Function `decrypt` (lines 96–150) receives the ciphertext to decrypt and the secret key and returns a string containing the original plaintext:

```

94    // decrypt receives the ciphertext and secret key, reverses
95    // Vigenère cipher process and returns the unencrypted plaintext
96    std::string decrypt(
97        std::string_view ciphertext, std::string_view secret) {
98
99        checkKey(secret); // ensure secret key contains only letters
100
101        std::string plaintext{}; // stores unencrypted text
102        size_t keyIndex{0}; // current letter index in secret key
103
104        for (size_t i = 0; i < ciphertext.size(); ++i) {
105            char c = ciphertext[i];
106            char s = secret[keyIndex];
107            char p = (c - s + 26) % 26;
108            plaintext += p;
109            keyIndex = (keyIndex + 1) % secret.size();
110        }
111
112        return plaintext;
113    }
114
115    void checkKey(const std::string &key) {
116        for (char c : key) {
117            if (!isalpha(c)) {
118                throw std::invalid_argument("Secret key must contain only letters");
119            }
120        }
121    }
122
123    std::string_view toLower(const std::string &s) {
124        std::string result;
125        for (char c : s) {
126            if (isupper(c)) {
127                result += tolower(c);
128            } else {
129                result += c;
130            }
131        }
132        return result;
133    }
134
135    std::string_view toUpper(const std::string &s) {
136        std::string result;
137        for (char c : s) {
138            if (islower(c)) {
139                result += toupper(c);
140            } else {
141                result += c;
142            }
143        }
144        return result;
145    }
146
147    std::string_view removePunctuation(const std::string &s) {
148        std::string result;
149        for (char c : s) {
150            if (isalpha(c)) {
151                result += c;
152            }
153        }
154        return result;
155    }
156
157    std::string_view removeSpaces(const std::string &s) {
158        std::string result;
159        for (char c : s) {
160            if (c != ' ') {
161                result += c;
162            }
163        }
164        return result;
165    }
166
167    std::string_view removeNonLetters(const std::string &s) {
168        std::string result;
169        for (char c : s) {
170            if (isalpha(c)) {
171                result += c;
172            }
173        }
174        return result;
175    }
176
177    std::string_view removeNonPrintable(const std::string &s) {
178        std::string result;
179        for (char c : s) {
180            if (isprint(c)) {
181                result += c;
182            }
183        }
184        return result;
185    }
186
187    std::string_view removeNonASCII(const std::string &s) {
188        std::string result;
189        for (char c : s) {
190            if (isascii(c)) {
191                result += c;
192            }
193        }
194        return result;
195    }
196
197    std::string_view removeNonLatin1(const std::string &s) {
198        std::string result;
199        for (char c : s) {
200            if (islatin1(c)) {
201                result += c;
202            }
203        }
204        return result;
205    }
206
207    std::string_view removeNonLatin2(const std::string &s) {
208        std::string result;
209        for (char c : s) {
210            if (islatin2(c)) {
211                result += c;
212            }
213        }
214        return result;
215    }
216
217    std::string_view removeNonLatin5(const std::string &s) {
218        std::string result;
219        for (char c : s) {
220            if (islatin5(c)) {
221                result += c;
222            }
223        }
224        return result;
225    }
226
227    std::string_view removeNonLatin9(const std::string &s) {
228        std::string result;
229        for (char c : s) {
230            if (islatin9(c)) {
231                result += c;
232            }
233        }
234        return result;
235    }
236
237    std::string_view removeNonLatin12(const std::string &s) {
238        std::string result;
239        for (char c : s) {
240            if (islatin12(c)) {
241                result += c;
242            }
243        }
244        return result;
245    }
246
247    std::string_view removeNonLatin15(const std::string &s) {
248        std::string result;
249        for (char c : s) {
250            if (islatin15(c)) {
251                result += c;
252            }
253        }
254        return result;
255    }
256
257    std::string_view removeNonLatin16(const std::string &s) {
258        std::string result;
259        for (char c : s) {
260            if (islatin16(c)) {
261                result += c;
262            }
263        }
264        return result;
265    }
266
267    std::string_view removeNonLatin20(const std::string &s) {
268        std::string result;
269        for (char c : s) {
270            if (islatin20(c)) {
271                result += c;
272            }
273        }
274        return result;
275    }
276
277    std::string_view removeNonLatin21(const std::string &s) {
278        std::string result;
279        for (char c : s) {
280            if (islatin21(c)) {
281                result += c;
282            }
283        }
284        return result;
285    }
286
287    std::string_view removeNonLatin22(const std::string &s) {
288        std::string result;
289        for (char c : s) {
290            if (islatin22(c)) {
291                result += c;
292            }
293        }
294        return result;
295    }
296
297    std::string_view removeNonLatin23(const std::string &s) {
298        std::string result;
299        for (char c : s) {
300            if (islatin23(c)) {
301                result += c;
302            }
303        }
304        return result;
305    }
306
307    std::string_view removeNonLatin24(const std::string &s) {
308        std::string result;
309        for (char c : s) {
310            if (islatin24(c)) {
311                result += c;
312            }
313        }
314        return result;
315    }
316
317    std::string_view removeNonLatin25(const std::string &s) {
318        std::string result;
319        for (char c : s) {
320            if (islatin25(c)) {
321                result += c;
322            }
323        }
324        return result;
325    }
326
327    std::string_view removeNonLatin26(const std::string &s) {
328        std::string result;
329        for (char c : s) {
330            if (islatin26(c)) {
331                result += c;
332            }
333        }
334        return result;
335    }
336
337    std::string_view removeNonLatin27(const std::string &s) {
338        std::string result;
339        for (char c : s) {
340            if (islatin27(c)) {
341                result += c;
342            }
343        }
344        return result;
345    }
346
347    std::string_view removeNonLatin28(const std::string &s) {
348        std::string result;
349        for (char c : s) {
350            if (islatin28(c)) {
351                result += c;
352            }
353        }
354        return result;
355    }
356
357    std::string_view removeNonLatin29(const std::string &s) {
358        std::string result;
359        for (char c : s) {
360            if (islatin29(c)) {
361                result += c;
362            }
363        }
364        return result;
365    }
366
367    std::string_view removeNonLatin30(const std::string &s) {
368        std::string result;
369        for (char c : s) {
370            if (islatin30(c)) {
371                result += c;
372            }
373        }
374        return result;
375    }
376
377    std::string_view removeNonLatin31(const std::string &s) {
378        std::string result;
379        for (char c : s) {
380            if (islatin31(c)) {
381                result += c;
382            }
383        }
384        return result;
385    }
386
387    std::string_view removeNonLatin32(const std::string &s) {
388        std::string result;
389        for (char c : s) {
390            if (islatin32(c)) {
391                result += c;
392            }
393        }
394        return result;
395    }
396
397    std::string_view removeNonLatin33(const std::string &s) {
398        std::string result;
399        for (char c : s) {
400            if (islatin33(c)) {
401                result += c;
402            }
403        }
404        return result;
405    }
406
407    std::string_view removeNonLatin34(const std::string &s) {
408        std::string result;
409        for (char c : s) {
410            if (islatin34(c)) {
411                result += c;
412            }
413        }
414        return result;
415    }
416
417    std::string_view removeNonLatin35(const std::string &s) {
418        std::string result;
419        for (char c : s) {
420            if (islatin35(c)) {
421                result += c;
422            }
423        }
424        return result;
425    }
426
427    std::string_view removeNonLatin36(const std::string &s) {
428        std::string result;
429        for (char c : s) {
430            if (islatin36(c)) {
431                result += c;
432            }
433        }
434        return result;
435    }
436
437    std::string_view removeNonLatin37(const std::string &s) {
438        std::string result;
439        for (char c : s) {
440            if (islatin37(c)) {
441                result += c;
442            }
443        }
444        return result;
445    }
446
447    std::string_view removeNonLatin38(const std::string &s) {
448        std::string result;
449        for (char c : s) {
450            if (islatin38(c)) {
451                result += c;
452            }
453        }
454        return result;
455    }
456
457    std::string_view removeNonLatin39(const std::string &s) {
458        std::string result;
459        for (char c : s) {
460            if (islatin39(c)) {
461                result += c;
462            }
463        }
464        return result;
465    }
466
467    std::string_view removeNonLatin40(const std::string &s) {
468        std::string result;
469        for (char c : s) {
470            if (islatin40(c)) {
471                result += c;
472            }
473        }
474        return result;
475    }
476
477    std::string_view removeNonLatin41(const std::string &s) {
478        std::string result;
479        for (char c : s) {
480            if (islatin41(c)) {
481                result += c;
482            }
483        }
484        return result;
485    }
486
487    std::string_view removeNonLatin42(const std::string &s) {
488        std::string result;
489        for (char c : s) {
490            if (islatin42(c)) {
491                result += c;
492            }
493        }
494        return result;
495    }
496
497    std::string_view removeNonLatin43(const std::string &s) {
498        std::string result;
499        for (char c : s) {
500            if (islatin43(c)) {
501                result += c;
502            }
503        }
504        return result;
505    }
506
507    std::string_view removeNonLatin44(const std::string &s) {
508        std::string result;
509        for (char c : s) {
510            if (islatin44(c)) {
511                result += c;
512            }
513        }
514        return result;
515    }
516
517    std::string_view removeNonLatin45(const std::string &s) {
518        std::string result;
519        for (char c : s) {
520            if (islatin45(c)) {
521                result += c;
522            }
523        }
524        return result;
525    }
526
527    std::string_view removeNonLatin46(const std::string &s) {
528        std::string result;
529        for (char c : s) {
530            if (islatin46(c)) {
531                result += c;
532            }
533        }
534        return result;
535    }
536
537    std::string_view removeNonLatin47(const std::string &s) {
538        std::string result;
539        for (char c : s) {
540            if (islatin47(c)) {
541                result += c;
542            }
543        }
544        return result;
545    }
546
547    std::string_view removeNonLatin48(const std::string &s) {
548        std::string result;
549        for (char c : s) {
550            if (islatin48(c)) {
551                result += c;
552            }
553        }
554        return result;
555    }
556
557    std::string_view removeNonLatin49(const std::string &s) {
558        std::string result;
559        for (char c : s) {
560            if (islatin49(c)) {
561                result += c;
562            }
563        }
564        return result;
565    }
566
567    std::string_view removeNonLatin50(const std::string &s) {
568        std::string result;
569        for (char c : s) {
570            if (islatin50(c)) {
571                result += c;
572            }
573        }
574        return result;
575    }
576
577    std::string_view removeNonLatin51(const std::string &s) {
578        std::string result;
579        for (char c : s) {
580            if (islatin51(c)) {
581                result += c;
582            }
583        }
584        return result;
585    }
586
587    std::string_view removeNonLatin52(const std::string &s) {
588        std::string result;
589        for (char c : s) {
590            if (islatin52(c)) {
591                result += c;
592            }
593        }
594        return result;
595    }
596
597    std::string_view removeNonLatin53(const std::string &s) {
598        std::string result;
599        for (char c : s) {
600            if (islatin53(c)) {
601                result += c;
602            }
603        }
604        return result;
605    }
606
607    std::string_view removeNonLatin54(const std::string &s) {
608        std::string result;
609        for (char c : s) {
610            if (islatin54(c)) {
611                result += c;
612            }
613        }
614        return result;
615    }
616
617    std::string_view removeNonLatin55(const std::string &s) {
618        std::string result;
619        for (char c : s) {
620            if (islatin55(c)) {
621                result += c;
622            }
623        }
624        return result;
625    }
626
627    std::string_view removeNonLatin56(const std::string &s) {
628        std::string result;
629        for (char c : s) {
630            if (islatin56(c)) {
631                result += c;
632            }
633        }
634        return result;
635    }
636
637    std::string_view removeNonLatin57(const std::string &s) {
638        std::string result;
639        for (char c : s) {
640            if (islatin57(c)) {
641                result += c;
642            }
643        }
644        return result;
645    }
646
647    std::string_view removeNonLatin58(const std::string &s) {
648        std::string result;
649        for (char c : s) {
650            if (islatin58(c)) {
651                result += c;
652            }
653        }
654        return result;
655    }
656
657    std::string_view removeNonLatin59(const std::string &s) {
658        std::string result;
659        for (char c : s) {
660            if (islatin59(c)) {
661                result += c;
662            }
663        }
664        return result;
665    }
666
667    std::string_view removeNonLatin60(const std::string &s) {
668        std::string result;
669        for (char c : s) {
670            if (islatin60(c)) {
671                result += c;
672            }
673        }
674        return result;
675    }
676
677    std::string_view removeNonLatin61(const std::string &s) {
678        std::string result;
679        for (char c : s) {
680            if (islatin61(c)) {
681                result += c;
682            }
683        }
684        return result;
685    }
686
687    std::string_view removeNonLatin62(const std::string &s) {
688        std::string result;
689        for (char c : s) {
690            if (islatin62(c)) {
691                result += c;
692            }
693        }
694        return result;
695    }
696
697    std::string_view removeNonLatin63(const std::string &s) {
698        std::string result;
699        for (char c : s) {
700            if (islatin63(c)) {
701                result += c;
702            }
703        }
704        return result;
705    }
706
707    std::string_view removeNonLatin64(const std::string &s) {
708        std::string result;
709        for (char c : s) {
710            if (islatin64(c)) {
711                result += c;
712            }
713        }
714        return result;
715    }
716
717    std::string_view removeNonLatin65(const std::string &s) {
718        std::string result;
719        for (char c : s) {
720            if (islatin65(c)) {
721                result += c;
722            }
723        }
724        return result;
725    }
726
727    std::string_view removeNonLatin66(const std::string &s) {
728        std::string result;
729        for (char c : s) {
730            if (islatin66(c)) {
731                result += c;
732            }
733        }
734        return result;
735    }
736
737    std::string_view removeNonLatin67(const std::string &s) {
738        std::string result;
739        for (char c : s) {
740            if (islatin67(c)) {
741                result += c;
742            }
743        }
744        return result;
745    }
746
747    std::string_view removeNonLatin68(const std::string &s) {
748        std::string result;
749        for (char c : s) {
750            if (islatin68(c)) {
751                result += c;
752            }
753        }
754        return result;
755    }
756
757    std::string_view removeNonLatin69(const std::string &s) {
758        std::string result;
759        for (char c : s) {
760            if (islatin69(c)) {
761                result += c;
762            }
763        }
764        return result;
765    }
766
767    std::string_view removeNonLatin70(const std::string &s) {
768        std::string result;
769        for (char c : s) {
770            if (islatin70(c)) {
771                result += c;
772            }
773        }
774        return result;
775    }
776
777    std::string_view removeNonLatin71(const std::string &s) {
778        std::string result;
779        for (char c : s) {
780            if (islatin71(c)) {
781                result += c;
782            }
783        }
784        return result;
785    }
786
787    std::string_view removeNonLatin72(const std::string &s) {
788        std::string result;
789        for (char c : s) {
790            if (islatin72(c)) {
791                result += c;
792            }
793        }
794        return result;
795    }
796
797    std::string_view removeNonLatin73(const std::string &s) {
798        std::string result;
799        for (char c : s) {
800            if (islatin73(c)) {
801                result += c;
802            }
803        }
804        return result;
805    }
806
807    std::string_view removeNonLatin74(const std::string &s) {
808        std::string result;
809        for (char c : s) {
810            if (islatin74(c)) {
811                result += c;
812            }
813        }
814        return result;
815    }
816
817    std::string_view removeNonLatin75(const std::string &s) {
818        std::string result;
819        for (char c : s) {
820            if (islatin75(c)) {
821                result += c;
822            }
823        }
824        return result;
825    }
826
827    std::string_view removeNonLatin76(const std::string &s) {
828        std::string result;
829        for (char c : s) {
830            if (islatin76(c)) {
831                result += c;
832            }
833        }
834        return result;
835    }
836
837    std::string_view removeNonLatin77(const std::string &s) {
838        std::string result;
839        for (char c : s) {
840            if (islatin77(c)) {
841                result += c;
842            }
843        }
844        return result;
845    }
846
847    std::string_view removeNonLatin78(const std::string &s) {
848        std::string result;
849        for (char c : s) {
850            if (islatin78(c)) {
851                result += c;
852            }
853        }
854        return result;
855    }
856
857    std::string_view removeNonLatin79(const std::string &s) {
858        std::string result;
859        for (char c : s) {
860            if (islatin79(c)) {
861                result += c;
862            }
863        }
864        return result;
86
```

```
104     for (size_t i{0}; i < ciphertext.size(); ++i) {
105         // determine whether character at i is lowercase so we can
106         // place a corresponding lowercase letter in plainText
107         const bool lower{std::islower(ciphertext.at(i)) ? true : false};
108
109         // convert currentChar to uppercase; for uppercase
110         // letters and non-letters the character remains the same
111         const char currentChar{gsl::narrow_cast<const char>(
112             std::toupper(ciphertext.at(i)))};
113
114         // if current is a letter decrypt it
115         if ('A' <= currentChar && currentChar <= 'Z') {
116             // to get the row index in the Vigenère square, select the
117             // character at keyIndex in the secret key, convert it to
118             // uppercase and subtract 'A' from it
119             const int row{std::toupper(secret.at(keyIndex)) - 'A'};
120
121             // increment the keyIndex, ensuring that it is
122             // reset to 0 if keyIndex reaches secret.size()
123             keyIndex = (keyIndex + 1) % secret.size();
124
125             // column in the Vigenère square
126             int column{-1};
127
128             // find currentChar's column in Vigenère square's current row
129             for (int i{0}; i < m_square.at(row).size(); ++i) {
130                 if (m_square.at(row).at(i) == currentChar) {
131                     column = i;
132                     break;
133                 }
134             }
135
136             // determine original character
137             const char originalChar{
138                 gsl::narrow_cast<const char>('A' + column)};
139
140             // add originalChar to plaintext in the correct case
141             plaintext +=
142                 (lower ? std::tolower(originalChar) : originalChar);
143             }
144             else {
145                 plaintext += currentChar; // add non-letter to plaintext
146             }
147         }
148
149         return plaintext; // return the unencrypted text
150     }
```

The function operates as follows:

- Line 99 calls utility function `checkKey` (defined lines 160–170) to ensure that the secret key contains only letters.
- Line 101 defines the string `plaintext`, which will store the unencrypted text.
- Line 102 defines the `size_t` `keyIndex`, which will keep track of the current character in the secret key. That character is used to select a Vigenère square row.
- For each character in `plaintext`, lines 104–147 either decrypt the letter and add it to `plaintext` or, if the character is not a letter, simply add it to `plaintext`.
- Line 107 determines whether the original character was lowercase, so we can maintain that case as we create the `plaintext`.
- Lines 111–112 convert the current character to uppercase. If it is already uppercase or is not a letter, `std::toupper` returns the original character. We convert `std::toupper`'s result (an `int`) to a `const char` and place it in `currentChar`.
- If `currentChar` is a letter (line 115), then lines 119–142 decrypt it.
- Line 119 converts the `secret`'s letter at `keyIndex` to uppercase, then subtracts '`A`' to produce a value from 0–26 representing the Vigenère square row to select.
- Line 123 increments the `keyIndex` to the next position in `secret`, setting it back to 0 if we've reached the end of `secret`.
- Lines 126–138 perform the Vigenère cipher decryption algorithm for one character. We determine the Vigenère square `column` index by iterating through the current `row` to locate the index (`i`) of the character that matches `currentChar`. Then, lines 137–138 add that column to '`A`' to determine `originalChar`.
- Lines 141–142 then append `originalChar` to `plaintext` using `currentChar`'s original case.
- If `currentChar` was not a letter, line 145 simply appends it to `plaintext`.

private Data and Utility Function `checkKey`

Line 153 defines the static constant `m_size`, representing the number of rows and columns in the Vigenère square. Line 156 defines the two-dimensional array of `char`s `m_square`, which the constructor (lines 16–39) populates with the Vigenère square. The static utility function `checkKey` (lines 160–170) receives a secret-key string and ensures that it contains only letters; otherwise, the function throws an `invalid_argument` exception because the key cannot be used with our Vigenère cipher implementation.

```
151 private:  
152     // number of rows and columns in the Vigenère square  
153     static constexpr size_t m_size{26};  
154  
155     // 26-by-26 array of characters to store the Vigenère square  
156     std::array<std::array<char, m_size>, m_size> m_square;  
157  
158     // utility function checks that secret key contains only letters;  
159     // throws an invalid_argument exception if key contains non-letters  
160     static void checkKey(std::string_view secret) {  
161         for (size_t i{0}; i < secret.size(); ++i) {  
162             // if the uppercase version of the character at index i  
163             // is not a letter throw an invalid_argument exception  
164             if (std::toupper(secret.at(i)) < 'A' ||  
165                 std::toupper(secret.at(i)) > 'Z') {  
166                 throw std::invalid_argument(  
167                     "key must contain only letters A-Z or a-z");  
168             }  
169         }  
170     }  
171 };
```

Weakness in Secret-Key Cryptography: A Look to Public-Key Cryptography

Secret-key encryption and decryption have a weakness—the ciphertext is only as secure as the secret key. The ciphertext can be decrypted by anyone who discovers or steals the secret key. **Public-key cryptography** performs encryption with a public key known to every sender who may want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. The messages can be decrypted only with a paired private key known only to the receiver, so it's much more secure than the secret key in secret-key cryptography. One of the most popular public-key cryptography schemes is RSA.^{41,42}

A Note about Cryptography and Computing Power

Ideally, Ciphertext should be impossible to “break”—that is, it should not be possible to determine the plaintext from the ciphertext without the decryption key. For various reasons, that goal is impractical. So, designers of cryptography schemes settle for making them extraordinarily difficult to break. One problem with today's increasingly powerful computers is that they're making it possible to break most encryption schemes in use over the last few decades.

41. “RSA (cryptosystem).” Accessed April 14, 2023. [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)).

42. “RSA Algorithm.” Accessed April 14, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.

Cryptography is at the root of cryptocurrencies such as Bitcoin.⁴³ The phenomenally powerful computers that quantum computing will make possible are putting cryptography schemes and cryptocurrencies at risk.^{44,45} The cryptocurrency community is working on these challenges.^{46,47,48}

9.23 Wrap-Up

In this chapter, you created your own classes, instantiated 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 initialize an object's data members.

We used a `Time` class case study to introduce various additional features. We showed how to separate a class's 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. 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

-
- 43. "Cryptocurrency." Accessed April 14, 2023. <https://www.investopedia.com/terms/c/cryptocurrency.asp>.
 - 44. "The Impact of Quantum Computing on Present Cryptography." Accessed April 14, 2023. <https://arxiv.org/pdf/1804.00200.pdf>.
 - 45. "Quantum Computing and its Impact on Cryptography." Accessed April 14, 2023. <https://www.cryptomathic.com/news-events/blog/quantum-computing-and-its-impact-on-cryptography>.
 - 46. "How Should Crypto Prepare for Google's 'Quantum Supremacy'?" Accessed April 14, 2023. <https://www.coindesk.com/how-should-crypto-prepare-for-googles-quantum-supremacy>.
 - 47. "Here's Why Quantum Computing Will Not Break Cryptocurrencies." Accessed April 14, 2023. <https://www.forbes.com/sites/rogerhuang/2020/12/21/heres-why-quantum-computing-will-not-break-cryptocurrencies/>.
 - 48. "How the Crypto World Is Preparing for Quantum Computing, Explained." Accessed April 14, 2023. <https://cointelegraph.com/explained/how-the-crypto-world-is-preparing-for-quantum-computing-explained>.

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 each non-static member function implicitly has a `this` pointer, allowing it 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 completed our Objects-Natural case study track by presenting our implementation of the Vigenère secret-key cipher, which you used to encrypt plaintext and decrypt ciphertext in Section 5.19's Objects Natural case study.

In the next chapter, we continue our discussion of classes by introducing inheritance. We'll see classes that share common attributes and behaviors can inherit them from a common "base" class. Then, we'll introduce polymorphism, which enables us to write programs that conveniently manipulate objects of classes related by inheritance, enabling us to conveniently "program in the general."

Exercises

9.1 (Class as a Blueprint) In our introduction to object orientation in Chapter 1, we mentioned that a class is like a blueprint. Explain why that's true in the context of Fig. 9.5's `Account` class.

9.2 (Default Constructor) What's a default constructor? How are an object's data members initialized if a class has only a default constructor defined by the compiler?

9.3 (static Data Members) State whether the following statement is true or false and, if false, explain why: Every object of a class has its own copy of all the class's data members.

9.4 (Set and Get Functions) Explain why a class might provide a `set` function and a `get` function for a data member.

9.5 (Modified Account Class) Modify class `Account` (Fig. 9.5) to provide a member function called `withdraw` that withdraws money from an `Account`. If the argument is negative the balance should remain unchanged. Modify class `AccountTest` (Fig. 9.6) to test member function `withdraw`.

9.6 (Square Class) Write a class that implements a `Square` shape. The class should contain a data member `m_side` property. Provide a constructor that takes an `int` side length as an argument. Provide a `getSide` function to return the side length. Also, provide the following member functions:

- `perimeter` returns $4 \times \text{side}$.
- `area` returns $\text{side} \times \text{side}$.
- `diagonal` returns the square root of the expression $(2 \times \text{side}^2)$.

The `perimeter`, `area` and `diagonal` functions should not have corresponding data attributes; rather, they should use `m_side` in calculations that return the desired values. Create a `Square` object and display its `m_side`, `perimeter`, `area` and `diagonal` values.

9.7 (Manipulating a `struct` Aggregate Type) Write a single statement or a set of statements to accomplish each of the following:

- a) Define an aggregate type `Part` containing an `int` variable `m_partNumber` and a `string m_partName`.
- b) Use separate statements to declare variable `part` to be of type `Part` and `parts` to be a `std::array` of 10 `Parts`.
- c) Read a part number and a part name from the keyboard into the members of variable `part`.
- d) Assign the member values of variable `part` to element 3 of array `parts`.
- e) Print the member values of element 3 of `parts`.

9.8 (*Invoice Class*) Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four data members—a part number (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`) and a price per item (type `int`). Your class should have a constructor that initializes the four data members. Provide a `set` and a `get` function for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an `int` value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0. Write a test program that demonstrates class `Invoice`'s capabilities.

9.9 (*Employee Class*) Create a class called `Employee` that includes three pieces of information as data members—a first name (type `string`), a last name (type `string`) and a monthly salary (type `int`). Your class should have a constructor that initializes the three data members. Provide a `set` and a `get` function for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10 percent raise and display each `Employee`'s yearly salary again.

9.10 (*Date Class*) Create a class called `Date` that includes three pieces of information as data members—a month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn't, set the month to 1. Provide a `set` and a `get` function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (/). Write a test program that demonstrates class `Date`'s capabilities.

9.11 (*Time Class Modification*) It would be perfectly reasonable for the `Time` class of Figs. 9.10–9.11 to represent the time internally as the number of seconds since midnight rather than the three integer values `m_hour`, `m_minute` and `m_second`. Clients could use the same `public` member functions and get the same results. Modify the `Time` class of Fig. 9.10 to implement the time as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class. [Note: This exercise nicely demonstrates the virtues of implementation hiding.]

9.12 (*Removing Duplicated Code in the main Function*) In Fig. 9.6, the `main` function contains six statements (lines 12–13, 14–15, 25–26, 27–28, 37–38 and 39–40) that each display an `Account` object's `m_name` and `m_balance`. Study these statements and you'll notice that they differ only in the `Account` object being manipulated—`account1` or `account2`. In this exercise, you'll define a new `displayAccount` function that contains one

copy of that output statement. The member function's parameter will be an `Account` object and the member function will output the object's name and balance. You'll then replace the six duplicated statements in `main` with calls to `displayAccount`, passing as an argument the specific `Account` object to output.

Modify Fig. 9.6 to define the following `displayAccount` function *before main*:

```
void displayAccount(const Account& account) {
    // place the statement that displays
    // account's name and balance here
}
```

Replace the comment in the member function's body with a statement that displays account's `m_name` and `m_balance`.

Once you've defined `displayAccount`'s, modify `main` to replace the statements that display each `Account`'s name and balance with calls to `displayAccount` of the form:

```
displayAccount(nameOfAccountObject);
```

Each call's argument should be the `account1` or `account2` object, as appropriate. Then, test the updated program to ensure that it produces the same output as shown in Fig. 9.6.

9.13 (Complex Class) Create a class called `Complex` for performing arithmetic with complex numbers. Write a program to test your class. Complex numbers have the form

$$\text{realPart} + \text{imaginaryPart} * i$$

where i is

$$\sqrt{-1}$$

Use `double` variables to represent the `private` data of the class. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided. Provide `public` member functions that perform the following tasks:

- a) `add`—Adds two `Complex` numbers: To form the real part of the result, add the real parts of the two `Complex` numbers. To form the imaginary part of the result, add the imaginary parts of two `Complex` numbers.
- b) `subtract`—Subtracts two `Complex` numbers: To form the real part of the result, subtract the real part of the `Complex` argument from the real part of the `Complex` object that calls `subtract`. To form the imaginary part of the result, subtract the imaginary part of the `Complex` argument from the imaginary part of the `Complex` object that calls `subtract`.
- c) `toString`—Returns a `string` representation of a `Complex` number in the form (a, b) , where a is the real part and b is the imaginary part.

In Chapter 11, you'll learn how to overload `+`, `-` and `<<` so you can write expressions like `a + b` and `a - b` and `cout << a` to add, subtract and output `Complex` objects. [Note: The C++ Standard Library provides its own class `complex` for complex-number arithmetic. For information on this class, visit <http://en.cppreference.com/w/cpp/numeric/complex>.]

9.14 (Standard Library Class `complex`) C++ supports complex numbers with the type `complex` from the `<complex>` header. Create the following `complex` numbers:

```
std::complex x{2, 4};
std::complex y{5, -1};
```

Then, use these `complex` numbers demonstrate:

- a) Adding two `complex` numbers with the `+` operator.
- b) Subtracting two `complex` numbers with the `-` operator.
- c) Printing `complex` numbers with `std::cout`.
- d) Passing a `complex` number to the function `std::real` to get the real part.
- e) Passing a `complex` number to the function `std::imag` to get the imaginary part.

9.15 (Composition: A Circle “Has a” Point at Its Center) A circle has a point at its center. Create a class `Point` that represents an (x,y) coordinate pair and provides:

- a) `private int` data members `m_x` and `m_y` with default values of 0.
- b) `public set` and `get` functions.
- c) A constructor receives x - and y -coordinate values to initialize a `Point`.
- d) A `move` function that receives x - and y -coordinate values and sets the `Point`'s new location
- e) A `toString` function that returns a `string` representation of a `Point`.

Create a class `Circle` that provides:

- a) A `private Point` data member `m_center`.
- b) A `private double` data member `m_radius`.
- c) A constructor that receives a `Point` representing the `Circle`'s center and a `double` representing the `Circle`'s radius.
- d) A `toString` function that returns a string representation of a `Circle`, including its center and radius.
- e) A `move` function that receives x - and y -coordinate values and sets a new location for the `Circle` by calling the composed `Point` object's `move` function.

Test your `Circle` class by creating a `Circle` object, displaying its string representation, moving the `Circle` and displaying its string representation again.

9.16 (Fraction Class) Create a class called `Fraction` for performing arithmetic with fractions. Write a program to test your class. Use integer variables to represent the `private` data of the class—the `m_numerator` and the `m_denominator`. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, the fraction

$$\frac{1}{2}$$

would be stored in the object as 1 in the `m_numerator` and 2 in the `m_denominator`. Provide `public` member functions that perform each of the following tasks:

- a) `add`—Adds two `Fractions`. The result should be stored in reduced form.
- b) `subtract`—Subtracts two `Fractions`. Store the result in reduced form.
- c) `multiply`—Multiplies two `Fractions`. Store the result in reduced form.
- d) `divide`—Divides two `Fractions`. The result should be stored in reduced form.

- e) `toString`—Returns a `string` representation of a `Fraction` in the form `a/b`, where `a` is the numerator and `b` is the denominator.
- f) `toDouble`—Returns the `Fraction` as a `double`.

In Chapter 11, you'll learn how to overload `+`, `-`, `*`, `/` and `<<` so you can write expressions like `a + b`, `a - b`, `a * b`, `a - b` and `cout << a` to add, subtract, multiply, divide and output `Fractions`.

9.17 (Enhancing Class Time) Modify the `Time` class of Figs. 9.10–9.11 to include a `tick` member function that increments the time stored in a `Time` object by one second. Write a program that tests the `tick` member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the `tick` member function works correctly. Be sure to test the following cases:

- a) Incrementing into the next minute.
- b) Incrementing into the next hour.
- c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

9.18 (Enhancing Class Date) Modify the `Date` class of Figs. 9.19–9.20 to perform error checking on the initializer values for data members `m_month`, `m_day` and `m_year`. Also, provide a member function `nextDay` to increment the day by one. Tests function `nextDay` in a loop that prints the date during each iteration to illustrate that `nextDay` works correctly. Be sure to test the following cases:

- a) Incrementing into the next month.
- b) Incrementing into the next year.

9.19 (Combining Class Time and Class Date) Combine the modified `Time` class of Exercise 9.17 and the modified `Date` class of Exercise 9.18 into one class called `DateAndTime`. Modify the `tick` function to call the `nextDay` function if the time increments into the next day. Modify functions `toStandardString` and `toUniversalString` so that each returns a `string` containing the date and time. Write a program to test the new class `DateAndTime`. Specifically, test incrementing the time into the next day.

9.20 (Returning Error Indicators from Class Time's set Functions) Modify the `set` functions in the `Time` class of Figs. 9.10–9.11 to return appropriate error values if an attempt is made to `set` a data member of an object of class `Time` to an invalid value. Write a program that tests your new version of class `Time`. Display error messages when `set` functions return error values.

9.21 (Rectangle Class) Create a class `Rectangle` with attributes `m_length` and `m_width`, each of which defaults to 1. Provide member functions that calculate the `perimeter` and the `area` of the rectangle. Also, provide `set` and `get` functions for the `m_length` and `m_width` attributes. The `set` functions should verify that `m_length` and `m_width` are each larger than 0.0 and less than 20.0.

9.22 (SavingsAccount Class) Create a `SavingsAccount` class. Use a `static` data member `m_annualInterestRate` to store the annual interest rate for each of the savers. Each account has a `private` data member `m_savingsBalance` indicating the amount the on deposit. Provide member function `calculateMonthlyInterest` that calculates the monthly interest by multiplying the `m_savingsBalance` by `m_annualInterestRate` divided by 12; this interest should be added to `m_savingsBalance`. Provide a `static` member function `modifyInterestRate` that sets the `static` `m_annualInterestRate` to a new value. Write

a driver program to test class `SavingsAccount`. Instantiate two different objects of class `SavingsAccount`, `saver1` and `saver2`, with balances of \$2000.00 and \$3000.00, respectively. Set `m_annualInterestRate` to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set `m_annualInterestRate` to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

9.23 (IntegerSet Class) Create class `IntegerSet` for which each object can hold integers in the range 0 through 100. Represent the set internally as a `vector` of `bool` values. Element `m_bools[i]` is `true` if integer `i` is in the set. Element `m_bools[j]` is `false` if integer `j` is not in the set. The default constructor initializes a set to the so-called “empty set,” i.e., a set for which all elements contain `false`.

- a) Provide member functions for the common set operations. For example, provide a `unionOfSets` member function that creates a third set that is the set-theoretic union of two existing sets (i.e., an element of the result is set to `true` if that element is `true` in either or both of the existing sets, and an element of the result is set to `false` if that element is `false` in each of the existing sets).
- b) Provide an `intersectionOfSets` member function which creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the result is set to `false` if that element is `false` in either or both of the existing sets, and an element of the result is set to `true` if that element is `true` in each of the existing sets).
- c) Provide an `insertElement` member function that places a new integer `k` into a set by setting `m_bools[k]` to `true`. Provide a `deleteElement` member function that deletes integer `m` by setting `m_bools[m]` to `false`.
- d) Provide a `toString` member function that returns a set as a string containing a list of numbers separated by spaces. Include only those elements that are present in the set (i.e., their position in the `vector` has a value of `true`). Return an empty string for an empty set.
- e) Provide an `isEqualTo` member function that determines whether two sets are equal.
- f) Provide an additional constructor that receives an array of integers, and uses the array to initialize a set object.

Now write a driver program to test your `IntegerSet` class. Instantiate several `IntegerSet` objects. Test that all your member functions work properly.

9.24 (Card Shuffling and Dealing) Create a program to shuffle and deal a deck of cards. The program should consist of class `Card`, class `DeckOfCards` and a driver program. Class `Card` should provide:

- a) Data members `m_face` and `m_suit`—use enumerations to represent the faces and suits.
- b) A constructor that receives two enumeration constants representing the face and suit and uses them to initialize the data members.
- c) Two static arrays of strings representing the faces and suits.
- d) A `toString` function that returns the `Card` as a string in the form “`face of suit`.” You can use the `+` operator to concatenate strings.

Class `DeckOfCards` should contain:

- a) An array of `Cards` named `m_deck` to store the `Cards`.

- b) An integer `currentCard` representing the next card to deal.
- c) A default constructor that initializes the `Cards` in the deck.
- d) A `shuffle` function that shuffles the `Cards` in the deck. The shuffle algorithm should iterate through the array of `Cards`. For each `Card`, randomly select another `Card` in the deck and swap the two `Cards`.
- e) A `dealCard` function that returns the next `Card` object from the deck.
- f) A `moreCards` function that returns a `bool` value indicating whether there are more `Cards` to deal.

The driver program should create a `DeckOfCards` object, shuffle the cards, then deal the 52 cards—the output should be similar to the following:

Six of Spades	Eight of Spades	Six of Clubs	Nine of Hearts
Queen of Hearts	Seven of Clubs	Nine of Spades	King of Hearts
Three of Diamonds	Deuce of Clubs	Ace of Hearts	Ten of Spades
Four of Spades	Ace of Clubs	Seven of Diamonds	Four of Hearts
Three of Clubs	Deuce of Hearts	Five of Spades	Jack of Diamonds
King of Clubs	Ten of Hearts	Three of Hearts	Six of Diamonds
Queen of Clubs	Eight of Diamonds	Deuce of Diamonds	Ten of Diamonds
Three of Spades	King of Diamonds	Nine of Clubs	Six of Hearts
Ace of Spades	Four of Diamonds	Seven of Hearts	Eight of Clubs
Deuce of Spades	Eight of Hearts	Five of Hearts	Queen of Spades
Jack of Hearts	Seven of Spades	Four of Clubs	Nine of Diamonds
Ace of Diamonds	Queen of Diamonds	Five of Clubs	King of Spades
Five of Diamonds	Ten of Clubs	Jack of Spades	Jack of Clubs

9.25 (Card Shuffling and Dealing) Modify the program you developed in Exercise 9.24 so that it deals a five-card poker hand. Then write functions to accomplish each of the following:

- a) Determine whether the hand contains a pair.
- b) Determine whether the hand contains two pairs.
- c) Determine whether the hand contains three of a kind (e.g., three jacks).
- d) Determine whether the hand contains four of a kind (e.g., four aces).
- e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).
- f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

9.26 (Project: Card Shuffling and Dealing) Use the functions from Exercise 9.25 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

9.27 (Project: Card Shuffling and Dealing) Modify the program you developed in Exercise 9.26 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand.

9.28 (Project: Card Shuffling and Dealing) Modify the program you developed in Exercise 9.27 so that it handles the dealer's hand, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and

determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. Play 20 more games. Does your modified program play a better game?

9.29 (*Project: Enhancing Class Time*) Investigate the capabilities of the C++ standard library's `<chrono>` header at <http://en.cppreference.com/w/cpp/chrono>. Then, provide a `Time` constructor that initializes a `Time` object using the current system time.

9.30 (*Project: TicTacToe Class*) Create a class `TicTacToe` that will enable you to write a complete program to play the game of tic-tac-toe (<https://en.wikipedia.org/wiki/Tic-tac-toe>). The class contains as private data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board. [Caution: This is an extremely challenging project that could take many weeks of effort!]

9.31 (*Project: Vigenère Cipher Modification—Supporting Digits*) Our Vigenère cipher implementation encrypts and decrypts only the letters A–Z. All other characters simply pass through as is. Modify the implementation to support the letters A–Z and the digits 0–9. Hint: This will require a 36-by-36 Vigenère square and the secret key may be composed of letters and digits.

Objects Natural Case Study Exercise: Serialization with JSON

9.32 (*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. In this case study exercise, we'll explore using JSON in C++.

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:

```
{"account": 100, "name": "Ito", "balance": 24.98}
```

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 "Ito"),
- 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 locally or for 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. Another common format is XML (eXtensible Markup Language).



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.”⁴⁹ 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.

cereal Header-Only Serialization Library

The **cereal header-only library**⁵⁰ 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

<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.

Serializing a vector of Objects Containing public Data

Let's begin by serializing objects containing only `public` data. In Fig. 9.38, we

- create a `vector` of `Record` objects and display its contents,
- use `cereal` to serialize it to a text file, then

49. “Deserialization Cheat Sheet.” OWASP Cheat Sheet Series. Accessed April 14, 2023. https://cheatsheetsseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

50. Copyright © 2017. W. Shane Grant and Randolph Voorhies, `cereal`—A C++11 library for serialization. <http://uscilab.github.io/cereal/>. All rights reserved.

- deserialize the file's contents into a `vector` of `Record` objects and display the deserialized `Records`.

To perform JSON serialization, include the `cereal` header `json.hpp` (line 3). To serialize a `std::vector`, include the `cereal` header `vector.hpp` (line 4).

```

1 // fig09_38.cpp
2 // Serializing and deserializing objects with the cereal library.
3 #include <cereal/archives/json.hpp>
4 #include <cereal/types/vector.hpp>
5 #include <format>
6 #include <fstream>
7 #include <iostream>
8 #include <vector>
9

```

Fig. 9.38 | Serializing and deserializing objects with the `cereal` library.

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.

```

10 struct Record {
11     int account{};
12     std::string first{};
13     std::string last{};
14     double balance{};
15 };
16

```

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 19–25) that receives an `Archive` as its first parameter and an object of your type as the second. 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.

```

17 // function template serialize is responsible for serializing and
18 // deserializing Record objects to/from the specified Archive
19 template <typename Archive>
20 void serialize(Archive& archive, Record& record) {
21     archive(cereal::make_nvp("account", record.account),
22             cereal::make_nvp("first", record.first),
23             cereal::make_nvp("last", record.last),
24             cereal::make_nvp("balance", record.balance));
25 }
26

```

Each `cereal` archive type has an overloaded parentheses operator that enables you to use an archive parameter as a function name, as in lines 21–24. 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` (short for “make name–value pair”), like line 21

```
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 and after deserialization. The function simply displays the contents of each Record in the vector it receives as an argument:

```
27 // display record at command line
28 void displayRecords(const std::vector<Record>& records) {
29     for (const auto& r : records) {
30         std::cout << std::format("{} {} {} {:.2f}\n",
31                               r.account, r.first, r.last, r.balance);
32     }
33 }
34
```

Creating Record Objects to Serialize

Lines 36–39 in `main` create a vector and initialize it with two Records (lines 37 and 38). The compiler uses class template argument deduction (CTAD) to determine the vector’s element type from the Records in the initializer list. Each Record is initialized with aggregate initialization. Line 42 outputs the vector’s contents to confirm that the two Records were initialized properly.

```
35 int main() {
36     std::vector records{
37         Record{100, "Bakary", "Zongo", 123.45},
38         Record{200, "Sofia", "Smirnova", 987.65}
39     };
40
41     std::cout << "Records to serialize:\n";
42     displayRecords(records);
43 }
```

```
Records to serialize:  
100 Bakary Zongo 123.45  
200 Sofia Smirnova 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 45 attempts to open the file `records.json` for writing. If it is successful, line 46 creates a `cereal::JSONOutputArchive` object named `archive` and initializes it with the `ofstream` object `output` so `archive` can write the JSON data into a file. Line 47 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 47 also results in one call to `serialize` (lines 19–25) for each `Record` object in the `vector`.

```
44     // serialize vector of Records to JSON and store in text file  
45     if (std::ofstream output{"records.json"}) {  
46         cereal::JSONOutputArchive archive{output};  
47         archive(cereal::make_nvp("records", records)); // serialize records  
48     }  
49 }
```

Contents of records.json

After line 47 executes, the file `records.json` contains the following JSON data:

```
{  
    "records": [  
        {  
            "account": 100,  
            "first": "Bakary",  
            "last": "Zongo",  
            "balance": 123.45  
        },  
        {  
            "account": 200,  
            "first": "Sofia",  
            "last": "Smirnova",  
            "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 47. 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.

```

50    // deserialize JSON from text file into vector of Records
51    if (std::ifstream input{"records.json"}) {
52        cereal::JSONInputArchive archive{input};
53        std::vector<Record> serializedRecords{};
54        archive(serializedRecords); // deserialize records
55        std::cout << "\nDeserialized records:\n";
56        displayRecords(serializedRecords);
57    }
58 }
```

```

Deserialized records:
100 Bakary Zongo 123.45
200 Sofia Smirnova 987.65
```

A `cereal::JSONInputArchive` deserializes JSON format from a specified stream. Line 51 attempts to open the file `records.json` for reading. If it is successful, line 52 creates the `cereal::JSONInputArchive` object `archive` and initializes it with the `ifstream` object `input` so `archive` can read JSON data from the `records.json` file. Line 53 creates an empty vector of `Records` into which we'll read the JSON data. Line 54 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 19–25), but because `archive` is a `cereal::JSONInputArchive`, each call to `serialize` reads one `Record`'s JSON data, creates a `Record` object and inserts the data into it.

9.33 (Project: 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, create a copy of Fig. 9.38 and replace the aggregate `Record` definition with the class `Record`. This `Record` class should contain `private` data, a constructor and `get` member functions that return each data member's value. The constructor's parameters should all have default arguments, allowing `cereal` to use this as the default constructor when deserializing `Record` objects. Use the following code in class `Record` to specify that the `serialize` function is a `friend` of the class:

```
template<typename Archive>
friend void serialize(Archive& archive, Record& record);
```

The `serialize` function now can access class `Record`'s `private` data members. The function `displayRecords` should use each `Record`'s `get` functions to access the data to display. The `main` function does not require changes and should produce the same results shown previously.



Security Case Study: Public-Key Cryptography

9.34 (RSA^{51,52,53} Public-Key Cryptography) In Section 9.22, you began learning about secret-key cryptography. The sender’s plaintext was encrypted with a secret key to form ciphertext. The receiver used the *same* secret key to decode the ciphertext, forming the original plaintext—this process is called **symmetric encryption**. A problem with secret-key cryptography is that the security of the ciphertext is only as good as the security of the secret key, and several copies of that key are “floating around.” In an attempt to correct this problem, public-key cryptography was proposed by Diffie–Hellman.⁵⁴

In this case-study exercise, we walk step-by-step through the **RSA Public-Key Cryptography algorithm**. In particular, we focus on how to generate:

- the **public key** that any sender can use to encrypt plaintext into ciphertext for a particular receiver, and
- the **private key** that only the particular receiver can use to decrypt the ciphertext.

RSA is based on sophisticated mathematics, but the steps you need to perform to generate the public and private keys, encrypt messages with the public key and decrypt messages with the private key are straightforward, as we’ll show momentarily. Industrial-quality RSA works with enormous prime numbers consisting of hundreds of digits. To keep our explanations simple and to enable you to quickly build a small-scale working version of RSA, we’re going to use only small prime numbers in our explanations. Such small-prime-number RSA versions are not very secure, but they’ll help you understand how RSA works.

Public-Key Cryptography

Whitfield Diffie and Martin Hellman, in their paper “New Directions in Cryptography,”⁵⁵ introduced **public-key cryptography** to address the weakness of secret-key cryptography—which is the vulnerability of the secret key having to be known by both the sender and the receiver. They came up with the idea but not an implementation of the scheme.

RSA Public-Key Cryptography

Rivest, Shamir and Adelman were the first to publish a working implementation of public-key cryptography. The scheme, called RSA⁵⁶, bears the initials of their last names. RSA is one of the most widely implemented public-key cryptography schemes in the world.⁵⁷

-
51. “RSA (cryptosystem).” Accessed April 14, 2023. [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)).
 52. “RSA Algorithm.” Accessed April 14, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.
 53. “PKCS #1: RSA Cryptography Specifications Version 2.2.” Accessed April 14, 2023. <https://tools.ietf.org/html/rfc8017>.
 54. “New Directions in Cryptography.” Accessed April 14, 2023. <https://ee.stanford.edu/~hellman/publications/24.pdf>.
 55. “New Directions in Cryptography.” Accessed April 14, 2023. <https://ee.stanford.edu/~hellman/publications/24.pdf>.
 56. R. Rivest; A. Shamir; L. Adleman (February 1978). “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” (PDF). Communications of the ACM. 21 (2): 120–126. <https://people.csail.mit.edu/rivest/Rsapaper.pdf>.
 57. “RSA algorithm (Rivest-Shamir-Adleman).” Accessed April 14, 2023. <https://searchsecurity.techtarget.com/definition/RSA>.

 Because RSA can be slow,⁵⁸ many organizations prefer to stick to faster private-key encryption, using RSA to securely send the secret key.

Historical Notes

Clifford Cocks in the U.K. created a workable public-key scheme several years before the RSA paper was published,⁵⁹ but his work was classified, so it was not revealed until about 20 years after RSA appeared.

The company RSA Security held a patent on the RSA algorithm. In 2000, that patent was coming due for renewal—instead of renewing, they placed the algorithm into the public domain.⁶⁰

RSA Algorithm Steps

Steps 1–5 below use small integer values to explain how the RSA algorithm generates a public-key/private-key pair. Then, Step 6 uses the public key to encrypt plaintext into ciphertext, and Step 7 uses the private key to decrypt the ciphertext back to the original plaintext. The steps we show are based on the original RSA paper⁶¹ and the RSA Algorithm Wikipedia page.⁶²

RSA Algorithm Step 1—Choose Two Prime Numbers

Choose two different prime numbers p and q . For this case study, we'll use small prime numbers— $p = 13$ and $q = 17$. This will keep the calculations manageable in our discussions and using C++'s limited-range, built-in integer data types. In commercial-grade RSA cryptography systems, these prime numbers typically are hundreds of digits each and chosen at random. For a sense of how large the integers in RSA can be, visit the RSA Numbers webpage

https://en.wikipedia.org/wiki/RSA_numbers

which shows various integers from 100 to 617 digits in length. The C++ integer data types `int`, `long int` and `long long int` cannot hold integers this large, so special processing is required to accommodate such large numbers, such as the arbitrary-sized integers in the Boost Multiprecision library.

RSA Algorithm Step 2—Calculate the Modulus (n), Which Is Part of Both the Public and Private Keys

Calculate the modulus n , which is simply the product of p and q :

$$n = p * q$$

Based on $p = 13$ and $q = 17$, n is 221. As you'll see, n is part of both the public and private keys. The p and q values are kept private.

-
- 58. “RSA (cryptosystem).” Accessed April 14, 2023. [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)).
 - 59. “Clifford Cocks.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Clifford_Cocks.
 - 60. “RSA Security Releases RSA Encryption Algorithm into Public Domain.” Accessed April 14, 2023. https://web.archive.org/web/20071120112201/http://www.rsa.com/press_release.aspx?id=261.
 - 61. R. Rivest; A. Shamir; L. Adleman (February 1978). “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” (PDF). Communications of the ACM. 21 (2): 120–126. <https://people.csail.mit.edu/rivest/Rsapaper.pdf>.
 - 62. “RSA Algorithm.” Accessed April 14, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.

RSA Algorithm Step 3—Calculate the Totient Function

Calculate $\Phi(n)$ —pronounced “phi of n”—which is **Euler’s totient function**.⁶³ This is calculated simply as:

$$\Phi(n) = (p - 1) * (q - 1)$$

Given $p = 13$ and $q = 17$, $\Phi(n)$ is

$$\Phi(n) = 12 * 16 = 192$$

This number is used in the calculations that determine the **encryption exponent (e)** and **decryption exponent (d)**, which will help us encrypt plaintext and decrypt ciphertext, respectively, as you’ll see below.

RSA Algorithm Step 4—Select the Public-Key Exponent (e) for Encryption Calculations

Next, we choose an **exponent, e**, for encryption, which is subject to the following rules:

- $1 < e < \Phi(n)$
- e must be **coprime** with $\Phi(n)$.

Two integers are **coprime** if they have no common factors other than 1.

In our example, the integers that satisfy the first rule for $\Phi(n) = 192$ are the values 2–191. The prime factorization of 192 is

$$192 = 2 * 2 * 2 * 2 * 2 * 2 * 3$$

The value for e must be coprime with $\Phi(n)$, so we must eliminate from consideration for e any prime factors and all their multiples. Thus, the value 2 and all the other even integers from 2–190 are eliminated, as are the value 3 and all its multiples. This leaves the following odd values as possible values for e:

5	7	11	13	17	19	23	25	29	31	35	37	41	43	47	49
53	55	59	61	65	67	71	73	77	79	83	85	89	91	95	97
101	103	107	109	113	115	119	121	125	127	131	133	137	139	143	145
149	151	155	157	161	163	167	169	173	175	179	181	185	187	191	

Any of these values can be used as the public encryption key’s exponent (e). For our continuing discussion we’ll choose 37, so our public key is (37, 221).

RSA Algorithm Step 5—Select the Private-Key Exponent (d) for Encryption Calculations

The final step is to determine the private key’s exponent, d, for decryption. We must choose a value for d such that

$$(d * e) \bmod \Phi(n) = 1$$

In our example, the first value of d for which this is true is 109. We can check whether the preceding calculation produces 1 by plugging in the values of d, e and $\Phi(n)$:

$$(109 * 37) \bmod 192$$

63. “Euler’s totient function.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Euler%27s_totient_function.

The value of $109 * 37$ is 4033. If you multiply 192 by 21, the result is 4032, leaving a remainder of 1. So, 109 is a valid value for d . There are many potential values of d —each is 109 plus a multiple of the totient (192). For instance, 301 ($109 + 1 * 192$):

$$(301 * 37) \bmod 192$$

$301 * 37$ is 11137, which has the remainder 1 when divided by 192— $192 * 58$ is 11136, leaving a remainder of 1. So values for d such as the following will work:

$$109 \quad 301 \quad 493 \quad 685 \quad 877 \quad \dots$$

We chose 109, so our private key is (109, 221).

Encrypting a Message with RSA

Once you have the public key, it's easy to encrypt a message using RSA. Given a plaintext integer message (M) to encrypt into ciphertext (C) and a public key consisting of two positive integers e (for encrypt) and n —commonly represented as (e, n) —a message sender can encrypt M with the calculation:

$$C = M^e \bmod n$$

The value of M must be in the range $0 \leq M < n$. Otherwise, you must break the message into values within that range and encrypt each separately.

Let's encrypt the M value 122 using our public key (37, 221):

$$C = 122^{37} \bmod 221$$

The value 122^{37} is an enormous number, but you can perform this calculation using the Wolfram Alpha website at

<https://www.wolframalpha.com/input/>

Enter the calculation as follows (the \wedge represents exponentiation in Wolfram Alpha):

$$122^{37} \bmod 221$$

You'll see that the result is 5, which is our ciphertext.

Decrypting a Message with RSA

It's also easy to decrypt a message if you have the private key. Given a ciphertext integer message (C) to decrypt into the original plaintext message (M) and a private key consisting of two positive integers d and n —commonly represented as (d, n) —a message receiver can decrypt C with the following calculation:

$$M = C^d \bmod n$$

Let's decrypt the C value 5 using our public key (109, 221):

$$C = 5^{109} \bmod 221$$

Once again, the value 5^{109} is an enormous number, but you can perform this calculation using Wolfram Alpha by entering the calculation as follows:

$$5^{109} \bmod 221$$

You'll see that the result is 122, which is our plaintext.

Note that n is part of *both* the public key and the private key. You'll also see that the exponent d 's value is based on the exponent e and the modulus value n .

Encrypting and Decrypting Strings

Suppose you wish to use RSA to encrypt a plaintext message, such as

Damn the torpedoes, full speed ahead!⁶⁴

As you know, the RSA algorithm encrypts *only* integer messages in the range $0 \leq M < n$. To encrypt the preceding message, you must map the characters to integer values.

One way to convert characters to integers is to use each character's numeric value in the underlying character set. For this exercise, assume ASCII characters, which have integer values in the range 0–127. Provided that a character's integer value is less than n , you can encrypt that value as shown previously. You can store each resulting ciphertext integer in a `vector<int>`. If you try to display those ciphertext integers as characters, you may see some strange symbols. For instance, the ciphertext integers may represent special characters, such as newlines or tabs, or may be outside the ASCII range. When you decrypt the ciphertext, you can take each resulting integer, cast it to a `char`, then append it to a `string` that will represent the deciphered plaintext.

Programming the RSA Algorithm

Now, implement the RSA algorithm in C++. Enable the user to encrypt and decrypt a simple integer, then encrypt and decrypt a line of text. Your program should produce an output dialog similar to the following:

```
Enter a prime number for p: 13
Enter a prime number for q: 17
n is 221
totient is 192
Candidates for e: 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 61
65 67 71 73 77 79 83 85 89 91 95 97 101 103 107 109 113 115 119 121 125 127
131 133 137 139 143 145 149 151 155 157 161 163 167 169 173 175 179 181 185
187 191
Select a value for e from the preceding candidates: 37
Candidate for d: 109
Select a value for d--either the d candidate above
or d plus a multiple of the totient: 109
Enter a non-negative integer less than n to encrypt: 122
The ciphertext is: 5
The decrypted plaintext is: 122
Enter a sentence to encrypt:
Damn the torpedoes, full speed ahead!
The ciphertext is:
DG`ue;X}eW;es9 fh s}eeW GueGW!
The decrypted plaintext is:
Damn the torpedoes, full speed ahead!
```

64. David Glasgow Farragut—an American Civil War Union officer and the first full admiral in the U.S. Navy. Accessed April 14, 2023. https://en.wikipedia.org/wiki/David_Farragut.

As you implement the RSA algorithm, keep the following hints in mind:

- **Modular exponentiation:** Raising a plaintext message to a large exponent (e.g., 122^{37}) results in enormous values that C++’s built-in integer types cannot represent. As you know, RSA encryption and decryption calculations perform both exponentiation and modulus operations. These can be combined using modular exponentiation to keep the RSA encryption and decryption calculations within manageable ranges. Define a function named `modularPow` that performs modular exponentiation. For the modular exponentiation algorithm, see the psuedocode at

<https://w.wiki/6bz9>

- **Calculating the greatest common divisor:** The candidate values for `e` (*Step 4*) must be coprime with the totient—again, their only common factor is 1. To determine if two numbers are coprime, you’ll need a function `gcd` that calculates the **greatest common divisor of two integers**. Your program should display all possible candidate values for `e`. You were asked to write a `gcd` function in Exercise 5.19.
- **Checking for prime numbers**—The RSA algorithm requires two prime numbers, `p` and `q`. Define a function `isPrime` to determine if an integer is prime. Use it to confirm that the `p` and `q` values the user entered are prime. Exercise 6.28 asked you to implement the Sieve of Eratosthenes to find prime values.

Your program also should define the following functions:

- A function to encrypt a plaintext message `M` using the public key (`e`, `n`):

```
int encrypt(int M, int e, int n);
```
- A function to decrypt a ciphertext message `C` using the private key (`d`, `n`):

```
int decrypt(int C, int d, int n);
```
- A function to encrypt a string by calling the `encrypt` function for each character of the string and placing the results into a `vector<int>`:

```
void encryptString(std::string_view plaintext,
                  std::vector<int>& ciphertext, int e, int n);
```
- A function to decrypt ciphertext from a `vector<int>` by calling the `decrypt` function for each integer. The function should return the plaintext `string`:

```
std::string decryptString(
    std::vector<int>& ciphertext, int d, int n);
```

References

For a video explanation of the RSA algorithm, see this two-part video presentation:

- The RSA Encryption Algorithm (1 of 2: Computing an Example):⁶⁵
<https://www.youtube.com/watch?v=4zahvcJ9g1g>
- The RSA Encryption Algorithm (2 of 2: Generating the Keys):⁶⁶
<https://www.youtube.com/watch?v=o0cTVTpUsPQ>

65. Woo, Eddie (misterwootube). “The RSA Encryption Algorithm (1 of 2: Computing an Example),” November 4, 2014. <https://www.youtube.com/watch?v=4zahvcJ9g1g>.

66. Woo, Eddie (misterwootube). “The RSA Encryption Algorithm (2 of 2: Generating the Keys),” November 4, 2014. <https://www.youtube.com/watch?v=o0cTVTpUsPQ>.

9.35 (An RSA Algorithm Improvement) In 1998, an improvement was made to the RSA algorithm replacing $\Phi(n)$ with $\lambda(n)$ (pronounced “lambda of n”):^{67,68}

$$\lambda(n) = \text{lcm}((p - 1), (q - 1))$$

where lcm represents the **least common multiple**.⁶⁹ We used $\Phi(n)$ in the previous RSA exercise with $p = 13$ and $q = 17$. The corresponding new $\lambda(n)$ calculation would be

$$\lambda(n) = \text{lcm}(12, 16)$$

where the least common multiple of 12 and 16 is 48, as you can see in the multiples below:

12	24	36	48	...
16	32	48	60	...

Make a copy of your code solution for the previous exercise and replace each use of $\Phi(n)$ with $\lambda(n)$, then test your updated code with the same prime-number values for p and q . When you encrypt the plaintext using the $\lambda(n)$ approach, your ciphertext will likely be different, but the decrypted plaintext should be the same.

9.36 (Stress Testing Your RSA Algorithm’s Limits) Try your program with gradually increasing values for p and q . How large do they get before the program no longer works with built-in data types? Also, test your program with increasingly larger candidates for e and d .

9.37 (Enhancing Your RSA Code) Modify your RSA program as follows:

- Your program displayed all the possible candidates for the encryption exponent e . Modify your program to show the first five potential values for the decryption exponent d (i.e., the first value of d plus $1 * \text{totient}$, the first value of d plus $2 * \text{totient}$, etc.). Follow your list of possibilities with an ellipsis (...).
- As your prime numbers p and q get larger, you’ll eventually surpass the `int` type’s maximum value limit. Modify your code to do all RSA integer calculations using type `cpp_int` class from the Boost Multiprecision library.

9.38 (Challenge Project: The RSA Problem⁷⁰) In this exercise, you’ll research attacks that have been perpetrated on industrial-strength RSA implementations. You’ll then try your own hand at cracking RSA ciphertext created by the small-scale RSA implementation you built in Exercise 9.34. Again, such small-scale implementations are not secure.

- Research the kinds of attacks that have been perpetrated against industrial-strength RSA systems. Note which kinds have succeeded and which have failed.
- RSA’s strength comes from the enormous prime numbers p and q (each typically hundreds of digits) used to calculate the far more enormous value of n (which is $p * q$) and the computational expense of factoring n to find p and q . The “RSA Problem” is the task of decrypting ciphertext given only the public key (e, n). This requires you to find n ’s prime factors p and q from which you would then derive d and decrypt the ciphertext.

Assume you have a public key (e, n) and ciphertext that was encrypted using that key with your small-scale RSA implementation, but you do not

67. “RSA Algorithm.” Accessed April 14, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.

68. “PKCS #1: RSA Cryptography Specifications, Version 2.0.” Accessed April 14, 2023. <https://tools.ietf.org/html/rfc2437>.

69. “Least common multiple.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/Least_common_multiple.

70. “RSA Problem.” Accessed April 14, 2023. https://en.wikipedia.org/wiki/RSA_problem.

know the private key required to decrypt the ciphertext. Use brute-force computing techniques to find n's prime factors p and q. Then, do the calculations necessary to recover d and decrypt the message.

9.39 (Challenging Research Project: Elliptic Curve Cryptography) [Note: Portions of this research project require sophisticated mathematics.] Exercise 9.34 presented public-key cryptography with RSA—the world's most widely used cryptographic system. RSA encryption could soon be broken due to rapidly increasing computing power and breakthroughs in prime-factorization algorithms.⁷¹

Elliptic Curve Cryptography (ECC)^{72,73}—based on complex mathematical properties of elliptic curves—is emerging as a powerful RSA alternative. It can achieve the same level of security with smaller keys—and extraordinary levels of security with larger ones. ECC uses significantly less processor bandwidth, storage and energy than RSA, so it's widely used for limited-resource systems, such as IoT devices, limited-battery-power mobile phones and cryptocurrency networks like Bitcoin and Ethereum. ECC is especially popular for securing exponentially growing Internet communications. Its lower energy requirements are appealing at a time when the enormous energy requirements of cryptocurrency networks are causing concern about their future growth and the growth of other blockchain-based solutions.

The mathematics of ECC is much more complex than that of RSA. Although that makes ECC systems more challenging to build and maintain, they are far more difficult to break than RSA. Public-key cryptography depends on finding functions that are easy to do in one direction but extremely difficult to do (time-wise) in reverse—these are called trapdoor functions. For example, in RSA, it's relatively fast and easy to multiply two large prime numbers to produce their non-prime product. However, it's extremely difficult and time-consuming to go in the reverse direction—finding a huge number's prime factors could take millions of years of computing time. By some estimates, breaking ECC systems could take time comparable to the length of the universe.⁷⁴

Research elliptic curve cryptography online and address each of the following:

- Explain how ECC uses elliptic curve equations of the following form to find public-key/private-key pairs:

$$y^2 = x^3 + ax + b$$

- Find a C++ ECC implementation on GitHub and study the code.
- Which elliptic curve does the implementation use?
- Explain the “easy” portion of the trapdoor function this implementation uses.
- Explain what the “hard” portion of this trapdoor function would require to break its ECC system.
- Briefly explain ECC’s known vulnerabilities.

71. Yan, Bao et al. “Factoring integers with sublinear resources on a superconducting quantum processor,” December 23, 2022. Accessed April 18, 2023. <https://arxiv.org/pdf/2212.12372.pdf>.

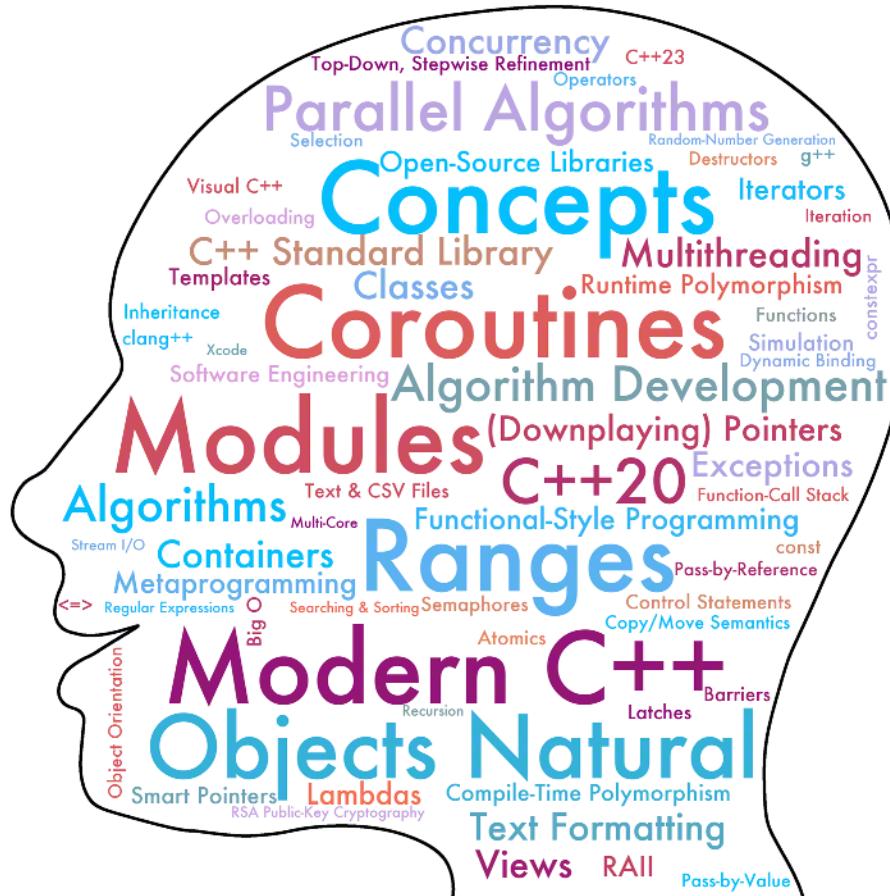
72. “Elliptic-curve cryptography.” Accessed April 18, 2023. https://en.wikipedia.org/wiki/Elliptic-curve_cryptography.

73. Matt Rickard, “Elliptic Curve Cryptography for Beginners.” March 27, 2022. Accessed April 18, 2023. <https://matt-rickard.com/elliptic-curve-cryptography>.

74. Ryan Sanders, “Elliptic Curve Cryptography: What is it? How does it work?” November 29, 2022. Accessed May 7, 2023. <https://www.keyfactor.com/blog/elliptic-curve-cryptography-what-is-it-how-does-it-work/>.

OOP: Inheritance and Runtime Polymorphism

10



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 get a sense of `virtual` function overhead.
 - Use interfaces to create more flexible runtime-polymorphic systems.

Outline

10.1	Introduction	10.7.5	<code>virtual</code> Destructors
10.2	Base Classes and Derived Classes	10.7.6	<code>final</code> Member Functions and Classes
10.2.1	CommunityMember Class Hierarchy	10.8	Abstract Classes and Pure <code>virtual</code> Functions
10.2.2	Shape Class Hierarchy and <code>public</code> Inheritance	10.8.1	Pure <code>virtual</code> Functions
10.3	Relationship Between Base and Derived Classes	10.8.2	Device Drivers: Polymorphism in Operating Systems
10.3.1	Creating and Using a SalariedEmployee Class	10.9	Case Study: Payroll System Using Runtime Polymorphism
10.3.2	Creating a SalariedEmployee–SalariedCommissionEmployee Inheritance Hierarchy	10.9.1	Creating Abstract Base Class Employee
10.4	Constructors and Destructors in Derived Classes	10.9.2	Creating Concrete Derived Class SalariedEmployee
10.5	Intro to Runtime Polymorphism: Polymorphic Video Game	10.9.3	Creating Concrete Derived Class CommissionEmployee
10.6	Relationships Among Objects in an Inheritance Hierarchy	10.9.4	Demonstrating Runtime Polymorphic Processing
10.6.1	Invoking Base-Class Functions from Derived-Class Objects	10.10	Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”
10.6.2	Aiming Derived-Class Pointers at Base-Class Objects	10.11	Program to an Interface, Not an Implementation
10.6.3	Derived-Class Member-Function Calls via Base-Class Pointers	10.11.1	Rethinking the Employee Hierarchy: CompensationModel Interface
10.7	Virtual Functions and Virtual Destructors	10.11.2	Class Employee
10.7.1	Why <code>virtual</code> Functions Are Useful	10.11.3	CompensationModel Implementations
10.7.2	Declaring <code>virtual</code> Functions	10.11.4	Testing the New Hierarchy
10.7.3	Invoking a <code>virtual</code> Function	10.11.5	Dependency Injection Design Benefits
10.7.4	<code>virtual</code> Functions in the SalariedEmployee Hierarchy	10.12	Wrap-Up Exercises

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 one or more objects of other classes as members. 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.

Runtime Polymorphism

We'll explain and demonstrate runtime polymorphism with inheritance hierarchies.¹ **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.

Implementing for Extensibility

With runtime polymorphism, you can design and implement more easily **extensible** systems. 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 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 the book, we’ll address current thinking and programming idioms.

Interfaces and Dependency Injection

To explain inheritance mechanics, 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, implementation 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 maintain and modify.

So, we’ll **refactor** one of our **implementation inheritance** 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—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

1. We’ll see later that runtime polymorphism does not have to be done with inheritance hierarchies, and we’ll also discuss compile-time polymorphism in Chapter 15, Templates, C++20 Concepts and Metaprogramming.

required by objects of the class—in our example, calculating an employee’s earnings. This pointer can be re-aimed—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.

Chapter Goal

Our goal in this chapter is to familiarize you with inheritance and runtime polymorphism mechanics, so you’ll be able to better appreciate modern **compile-time polymorphism** approaches that can promote ease of modifiability and better performance. You’ll see some of these approaches in Chapter 15, Templates, C++20 Concepts and Metaprogramming, and Chapter 20, Other Topics and a Look Toward the Future of C++. In Chapter 20, we’ll discuss

- non-virtual interface idiom (NVI),
- runtime polymorphism via `std::variant` and `std::visit`, and
- additional inheritance-related topics.



Checkpoint

1 (*Fill-in*) Runtime _____ enables programs to process objects of classes related by inheritance as if they’re all objects of the base-class type.

Answer: polymorphism.

2 (*Fill-in*) In _____ systems, 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.

Answer: extensible.

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

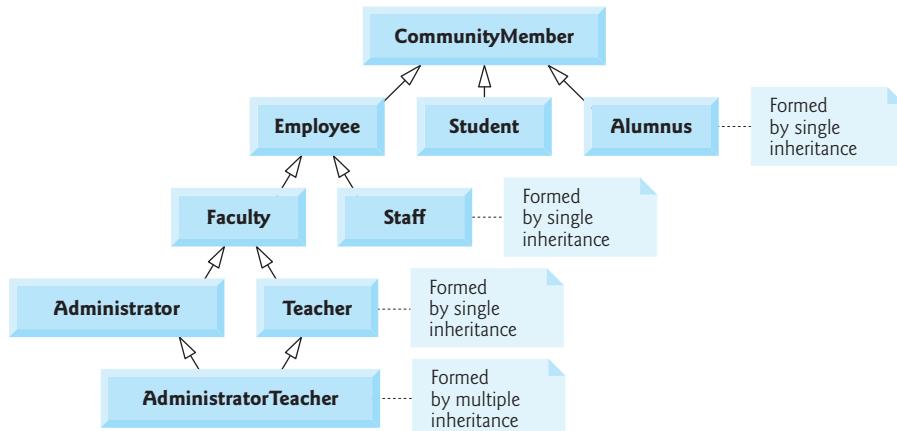
Every derived-class object *is an* object of its base-class type, and one base class can have many derived classes. So, the set of objects a base class represents is typically larger than the set of objects a derived class represents. For example, the base class `Vehicle` represents all vehicles, including cars, trucks, boats, airplanes, bicycles, etc. By contrast, the derived-class `Car` represents a smaller, more specific subset of all `Vehicle`s.

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.² A class can be

1. a base class that supplies members to other classes,
2. a derived class that inherits members from other classes or
3. both.³

Let's develop the 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. Following 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 also *is-a* **Teacher**), *is-a* **Faculty** member, *is-an* **Employee** and *is-a* **CommunityMember**.

-
2. We'll see that tightly coupled classes can make systems difficult to modify. We'll show alternatives that avoid tight coupling.
 3. 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>.



Checkpoint

1 (*True/False*) 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. A class can be a base class that supplies members to other classes, a derived class that inherits members from other classes or both.

Answer: True.

2 (*Fill-in*) With _____, a derived class inherits from one base class. With _____, a derived class inherits from two or more base classes.

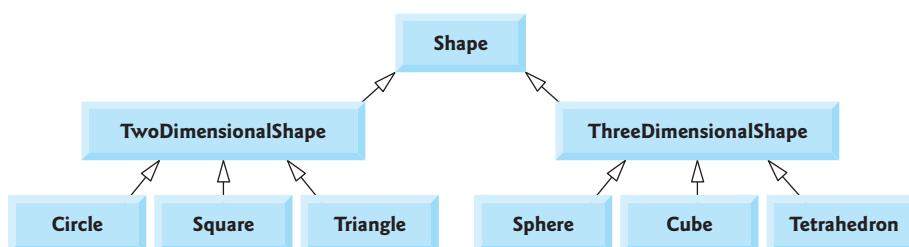
Answer: single inheritance, multiple inheritance.

3 (*Fill-in*) A(n) _____ base class is two or more levels up the class hierarchy from its derived classes.

Answer: indirect.

10.2.2 Shape Class Hierarchy and public Inheritance

Let's consider a Shape inheritance hierarchy:



Classes `TwoDimensionalShape` and `ThreeDimensionalShape` derive from base class `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`:

```
class TwoDimensionalShape : public Shape
```

This is **public inheritance**, which we'll use in this chapter. In **public inheritance**, **public** base-class members become **public** derived-class members and **protected** base-class members become **protected** derived-class members.⁴ Although **private** base-class members are not directly accessible in derived classes, 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 they provide such functionality. We'll discuss **private** and **protected** inheritance in Chapter 20, Other Topics and a Look Toward the Future of C++.



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

4. We discuss **protected** in Chapter 20

`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.



Checkpoint

1 (*True/False*) With `public` inheritance, 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.

Answer: True.

2 (*Fill-in*) The commonalities between base-class and derived-class objects are expressed in the _____.

Answer: base-class members.

3 (*Code*) Write the header of a class definition that would indicate class `Square` is a subclass of `TwoDimensionalShape`.

Answer: `class Square : public TwoDimensionalShape`

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 set` and `get` 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 22–28) validates its argument before modifying the data member `m_salary`.

```
1 // Fig. 10.1: SalariedEmployee.h
2 // SalariedEmployee class definition.
3 #pragma once // prevent multiple inclusions of header
```

Fig. 10.1 | `SalariedEmployee` class definition. (Part I of 2.)

```

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 };

```

Fig. 10.1 | SalariedEmployee class definition. (Part 2 of 2.)

```

1 // Fig. 10.2: SalariedEmployee.cpp
2 // Class SalariedEmployee member-function definitions.
3 #include <format>
4 #include <stdexcept>
5 #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7 // constructor
8 SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9     : m_name{name} {
10     setSalary(salary);
11 }
12
13 // set name
14 void SalariedEmployee::setName(std::string_view name) {
15     m_name = name; // should validate
16 }
17
18 // return name
19 std::string SalariedEmployee::getName() const {return m_name;}
20
21 // set salary
22 void SalariedEmployee::setSalary(double salary) {
23     if (salary < 0.0) {
24         throw std::invalid_argument("Salary must be >= 0.0");
25     }
26
27     m_salary = salary;
28 }
29

```

Fig. 10.2 | Class SalariedEmployee member-function definitions. (Part 1 of 2.)

```
30 // return salary
31 double SalariedEmployee::getSalary() const {return m_salary;}
32
33 // calculate earnings
34 double SalariedEmployee::earnings() const {return getSalary();}
35
36 // return string representation of SalariedEmployee object
37 std::string SalariedEmployee::toString() const {
38     return std::format("name: {}\nsalary: ${:.2f}\n", getName(),
39     getSalary());
40 }
```

Fig. 10.2 | Class SalariedEmployee member-function definitions. (Part 2 of 2.)

SalariedEmployee Constructor

The class's constructor (Fig. 10.2, lines 8–11) 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 34) calls `getSalary` and returns the result. Function `toString` (lines 37–40) returns a `string` containing a `SalariedEmployee` object's information.

Testing Class SalariedEmployee

Figure 10.3 tests class `SalariedEmployee`. Line 9 creates the `SalariedEmployee` object `employee`. Lines 12–14 demonstrate the `employee`'s `get` functions. Line 16 uses `setSalary` to change the `employee`'s `m_salary` value. Then, lines 17–18 call `employee`'s `toString` member function to get and output the `employee`'s updated information. Finally, line 21 displays the `employee`'s `earnings` using the updated `m_salary` value.

```
1 // fig10_03.cpp
2 // SalariedEmployee class test program.
3 #include <iostream>
4 #include <format>
5 #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7 int main() {
8     // instantiate a SalariedEmployee object
9     SalariedEmployee employee{"Sierra Dembo", 300.0};
10
11    // get SalariedEmployee data
12    std::cout << "Employee information obtained by get functions:\n"
13    << std::format("name: {}\nsalary: ${:.2f}\n", employee.getName(),
14    employee.getSalary());
15
16    employee.setSalary(500.0); // change salary
17    std::cout << "\nUpdated employee information from function toString:\n"
18    << employee.toString();
19}
```

Fig. 10.3 | SalariedEmployee class test program. (Part 1 of 2.)

```

20     // display only the employee's earnings
21     std::cout << std::format("\nearnings: {:.2f}\n", employee.earnings());
22 }

```

```

Employee information obtained by get functions:
name: Sierra Dembo
salary: $300.00

Updated employee information from function toString:
name: Sierra Dembo
salary: $500.00
earnings: $500.00

```

Fig. 10.3 | SalariedEmployee class test program. (Part 2 of 2.)

10.3.2 Creating a SalariedEmployee–SalariedCommissionEmployee Inheritance Hierarchy

Now, let's create a SalariedCommissionEmployee class (Figs. 10.4–10.5) that inherits from SalariedEmployee (Figs. 10.1–10.2). 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 *set* and *get* functions that manipulate the class's `m_grossSales` and `m_commissionRate` data members.

```

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 };

```

Fig. 10.4 | SalariedCommissionEmployee class definition indicating inheritance relationship with class CommissionEmployee.

Inheritance

The colon (:) in line 8 indicates inheritance, and public to its right specifies the kind of inheritance. In public inheritance, public base-class members remain public in the derived class and protected ones (discussed in Chapter 20) remain protected. The 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

A SE

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`.

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.

A SE

SalariedCommissionEmployee's Implementation

Figure 10.5 shows `SalariedCommissionEmployee`'s member-function implementations. Each derived-class constructor must call a base-class constructor. The `SalariedCommissionEmployee` constructor (lines 8–15) does this explicitly via a **base-class initializer** (line 11)—a member initializer that invokes the base-class constructor by name with the appropriate arguments. In this case, we initialize the inherited data members with the arguments `name` and `salary`. Functions `setGrossSales` (lines 18–24) and `setCommissionRate` (lines 32–41) validate their arguments before modifying the data members `m_grossSales` and `m_commissionRate`.

A SE

```
1 // Fig. 10.5: SalariedCommissionEmployee.cpp
2 // Class SalariedCommissionEmployee member-function definitions.
3 #include <iostream>
4 #include <stdexcept>
5 #include "SalariedCommissionEmployee.h"
6
```

Fig. 10.5 | Class `SalariedCommissionEmployee` member-function definitions. (Part 1 of 2.)

5. It's common in a derived-class constructor to simply pass its arguments to a corresponding base-class constructor and do nothing else. Chapter 20, Other Topics, shows how to inherit a base class's constructors.

```

7 // constructor
8 SalariedCommissionEmployee::SalariedCommissionEmployee(
9     std::string_view name, double salary, double grossSales,
10    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 std::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 // return commission rate
32 void SalariedCommissionEmployee::setCommissionRate(
33   double commissionRate) {
34
35   if (commissionRate <= 0.0 || commissionRate >= 1.0) {
36     throw std::invalid_argument(
37       "Commission rate must be > 0.0 and < 1.0");
38   }
39
40   m_commissionRate = commissionRate;
41 }
42
43 // get commission rate
44 double SalariedCommissionEmployee::getCommissionRate() const {
45   return m_commissionRate;
46 }
47
48 // calculate earnings--uses SalariedEmployee::earnings()
49 double SalariedCommissionEmployee::earnings() const {
50   return SalariedEmployee::earnings() +
51     getGrossSales() * getCommissionRate();
52 }
53
54 // returns string representation of SalariedCommissionEmployee object
55 std::string SalariedCommissionEmployee::toString() const {
56   return std::format(
57     "{}gross sales: {:.2f}\ncommission rate: {:.2f}\n",
58     SalariedEmployee::toString(), getGrossSales(), getCommissionRate());
59 }
```

Fig. 10.5 | Class SalariedCommissionEmployee member-function definitions. (Part 2 of 2.)

SalariedCommissionEmployee Member Function `earnings`

`SalariedCommissionEmployee`'s `earnings` function (lines 49–52) redefines `earnings` from class `SalariedEmployee` (Fig. 10.2, line 34) to calculate a `SalariedCommissionEmployee`'s earnings. Line 50 in `SalariedCommissionEmployee`'s version uses the expression

```
SalariedEmployee::earnings()
```

to get the portion of the earnings based on salary then adds that value to the commission to calculate the total earnings.

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, we avoid duplicating code by calling `SalariedEmployee`'s `earnings` function from `SalariedCommissionEmployee`'s `earnings` function.⁶

A SE

Err

SalariedCommissionEmployee Member Function `toString`

Similarly, `SalariedCommissionEmployee`'s `toString` function (Fig. 10.5, lines 55–59) redefines `SalariedEmployee`'s `toString` function (Fig. 10.2, lines 37–40). The new version returns a `string` containing

- the result of calling `SalariedEmployee::toString()` (Fig. 10.5, line 58) and
- the `SalariedCommissionEmployee`'s gross sales and commission rate.

Testing Class `SalariedCommissionEmployee`

Figure 10.6 creates the `SalariedCommissionEmployee` object `employee` (line 9). Lines 12–16 output the `employee`'s data by calling its `get` functions to retrieve the object's data member values. Lines 18–19 use `setGrossSales` and `setCommissionRate` to change the `employee`'s `m_grossSales` and `m_commissionRate` values, respectively. Then, lines 20–21 call `employee`'s `toString` member function to show the `employee`'s updated information. Finally, line 24 displays the `employee`'s updated earnings.

```

1 // fig10_06.cpp
2 // SalariedCommissionEmployee class test program.
3 #include <iostream>
4 #include <format>
5 #include "SalariedCommissionEmployee.h"
6
7 int main() {
8     // instantiate SalariedCommissionEmployee object
9     SalariedCommissionEmployee employee{"Ivano La1", 300.0, 5000.0, .04};
10
11    // get SalariedCommissionEmployee data
12    std::cout << "Employee information obtained by get functions:\n"
13    << std::format("{}: {}{}\n{}: {:.2f}\n{}: {:.2f}\n{}: {:.2f}\n",
14        "name", employee.getName(), "salary", employee.getSalary(),
15        "gross sales", employee.getGrossSales(),
16        "commission", employee.getCommissionRate());

```

Fig. 10.6 | `SalariedCommissionEmployee` class test program. (Part I of 2.)

6. Avoiding code duplication is the upside. The downside is that this creates a coupling between base and derived classes that can make large-scale systems difficult to modify.

```

17     employee.setGrossSales(8000.0); // change gross sales
18     employee.setCommissionRate(0.1); // change commission rate
19     std::cout << "\nUpdated employee information from function toString:\n"
20         << employee.toString();
21
22     // display the employee's earnings
23     std::cout << std::format("\nearnings: ${:.2f}\n", employee.earnings());
24 }

```

Employee information obtained by get functions:

```

name: Ivano Lal
salary: $300.00
gross sales: $5000.00
commission: 0.04

```

Updated employee information from function toString:

```

name: Ivano Lal
salary: $300.00
gross sales: $8000.00
commission rate: 0.10

```

earnings: \$1100.00

Fig. 10.6 | SalariedCommissionEmployee class test program. (Part 2 of 2.)

A Derived-Class Constructor Must Call Its Base Class's Constructor

The compiler would issue an error if SalariedCommissionEmployee's constructor did not explicitly invoke SalariedEmployee's constructor. In this case, the compiler would

Err attempt to call 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

SE 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 the base-class initializer(s) 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 directly by class SalariedCommissionEmployee's member functions. If they try, the compiler produces an

Err error message, such as the following from GNU g++:

```
'double SalariedEmployee::m_salary' is private within this context
```

However, as you saw in Fig. 10.5, SalariedCommissionEmployee's member functions can access the **public** members inherited from 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`.
- The compiler determines an object's size from its class's definition, so it must know the class definition to reserve the proper amount of memory for a new 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.
- 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 uses the base class's function prototypes to validate derived-class function calls to inherited base-class functions.

A SE

Eliminating Repeated Code via Implementation Inheritance

A SE

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



Checkpoint

- 1 *(Fill-in)* A derived-class constructor can call its base-class's constructor via a(n) _____—a member initializer that passes arguments to a base-class constructor.

Answer: base-class initializer.

- 2 *(Code)* Assume class `SalariedEmployee` has a `print` member function that takes no arguments. Also, assume class `SalariedCommissionEmployee` inherits from `SalariedEmployee` and redefines `print`. Write an expression that would call `SalariedEmployee`'s `print` function from `SalariedCommissionEmployee`'s `print` function.

Answer: `SalariedEmployee::print()`

- 3 *(True/False)* 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.

Answer: True.

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.
8. Again, the downside of implementation inheritance, especially in deep inheritance hierarchies, is that it creates a tight coupling among the hierarchy's classes, making it difficult to modify.

10.4 Constructors and Destructors in Derived Classes

Order of Constructor Calls



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**. This process continues for each level of the hierarchy until the constructor for the base class at the top of the hierarchy is called. 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 returns control to `SalariedCommissionEmployee`'s constructor, which initializes `m_grossSales` and `m_commissionRate`.

Order of Destructor Calls



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



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

- 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 order of their corresponding constructors.



Checkpoint

I (Fill-in) Instantiating a derived-class object begins a chain of constructor calls. The last constructor called in this chain is for the _____.

Answer: base class at the top of the hierarchy.

2 (*True/False*) 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.

Answer: True.

3 (*True/False*) Destructors for data members are called in the same order that their corresponding constructors were called.

Answer: False. Actually, destructors for data members are called in the reverse order that 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. The screen manager periodically sends each object the same `draw` message to refresh the screen. 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 the `draw` member function 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."

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 appropriately without knowing their types, as long as they belong to the same class hierarchy and are accessed via a base-class pointer or reference.⁹

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 be plugged into such a system without modifying the base system. Only code that instantiates new objects must be modified to accommodate new types.



9. In Chapters 15 and 20, we'll see that there are forms of compile-time polymorphism and runtime polymorphism that do not depend on inheritance hierarchies.



Checkpoint

1 (*True/False*) A polymorphic screen manager facilitates adding new classes to a system with minimal code modifications. New object types that can respond to existing messages can simply be plugged into such a system without modifying the base system. Only code that instantiates new objects must be modified to accommodate new types.

Answer: True.

2 (*True/False*) Runtime polymorphism enables you to deal in specifics and let the execution-time environment concern itself with the generalities.

Answer: False. Actually, runtime polymorphism enables you to deal in generalities and let the execution-time environment concern itself with the specifics.

10.6 Relationships Among Objects in an Inheritance Hierarchy

Let's examine the relationships among classes in an inheritance hierarchy more closely. The next several sections use Section 10.3's class hierarchy to present examples that aim base-class and derived-class pointers at base-class and derived-class objects, and show how those pointers can be used to invoke member functions that manipulate those objects:

Err

Err

- 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. **Compilation errors occur when we attempt to invoke derived-class-only member functions through the base-class pointer.**
- Finally, Section 10.7 introduces **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. **This is 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.

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.

SE

10.6.1 Invoking Base-Class Functions from Derived-Class Objects

Figure 10.7 reuses Section 10.3's `SalariedEmployee` and `SalariedCommissionEmployee` classes. 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.
- 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 available in the derived-class object**.

```
1 // fig10_07.cpp
2 // Aiming base-class and derived-class pointers at base-class
3 // and derived-class objects, respectively.
4 #include <format>
5 #include <iostream>
6 #include "SalariedEmployee.h"
7 #include "SalariedCommissionEmployee.h"
8
9 int main() {
10     // create base-class object
11     SalariedEmployee salaried{"Sierra Dembo", 500.0};
12
13     // create derived-class object
14     SalariedCommissionEmployee salariedCommission{
15         "Ivano Lal", 300.0, 5000.0, .04};
16
17     // output objects salaried and salariedCommission
18     std::cout << std::format("{}\n{}{}\n",
19         "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS",
20         salaried.toString(), // base-class toString
21         salariedCommission.toString()); // derived-class toString
22
23     // natural: aim base-class pointer at base-class object
24     SalariedEmployee* salariedPtr{&salaried};
25     std::cout << std::format("{}\n{}:{}\n{}\n",
26         "CALLING TOSTRING WITH BASE-CLASS POINTER TO",
27         "BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
28         salariedPtr->toString()); // base-class version
29
30     // natural: aim derived-class pointer at derived-class object
31     SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
32
33     std::cout << std::format("{}\n{}:{}\n{}\n",
34         "CALLING TOSTRING WITH DERIVED-CLASS POINTER TO",
35         "DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
36         salariedCommissionPtr->toString()); // derived-class version
```

Fig. 10.7 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 2.)

```

37      // aim base-class pointer at derived-class object
38      salariedPtr = &salariedCommission;
39      std::cout << std::format("{}\n{}:\n{}\n",
40          "CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS",
41          "OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
42          salariedPtr->toString()); // base class version
43
44  }

```

DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:

name: Sierra Dembo

salary: \$500.00

name: Ivano Lal

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: Sierra Dembo

salary: \$500.00

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO

DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:

name: Ivano Lal

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: Ivano Lal

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` who earns a commission based on gross sales. `SalariedCommissionEmployee`'s `earnings` member function (Fig. 10.5, lines 49–52) **redefines** `SalariedEmployee`'s version (Fig. 10.2, line 34) to include the commission. `SalariedCommissionEmployee`'s `toString` member function (Fig. 10.5, lines 55–59) **redefines** `SalariedEmployee`'s version (Fig. 10.2, lines 37–40) to return the same information and the employee's commission.

Creating Objects and Displaying Their Contents

In Fig. 10.7, line 11 creates a `SalariedEmployee` object and lines 14–15 create a `SalariedCommissionEmployee` object. Lines 20 and 21 use each object's name to invoke its `toString` member function.

Aiming a Base-Class Pointer at a Base-Class Object

Line 24 initializes the `SalariedEmployee` pointer `salariedPtr` with the base-class object `salaried`'s address. Line 28 uses the pointer to invoke the `salaried` object's `toString` member function from the base-class `SalariedEmployee`.

Aiming a Derived-Class Pointer at a Derived-Class Object

Line 31 initializes `SalariedCommissionEmployee` pointer `salariedCommissionPtr` with derived-class object `salariedCommission`'s address. Line 36 uses the pointer to invoke the `salariedCommission` object's `toString` member function from the derived-class `SalariedCommissionEmployee`.

Aiming a Base-Class Pointer at a Derived-Class Object

Line 39 assigns the 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 43 uses this pointer to invoke `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 **virtual functions make it possible to invoke the object type's functionality**—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`.

```
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{"Sierra Dembo", 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 *'
```

Fig. 10.8 | Aiming a derived-class pointer at a base-class object.

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

Err  neither `setGrossSales` and `setCommissionRate` member functions nor data members `m_grossSales` and `m_commissionRate` to set. If this were allowed, it 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.

10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers

Err  The compiler allows us to invoke only base-class member functions via a base-class pointer. So, a compilation error occurs 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. Figure 10.9 shows the compiler errors for invoking a derived-class-only member function via a base-class pointer (lines 22–24).

```

1 // fig10_09.cpp
2 // Attempting to call derived-class-only functions
3 // via a base-class pointer.
4 #include <iostream>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7
8 int main() {
9     SalariedCommissionEmployee salariedCommission{
10         "Ivano Lal", 300.0, 5000.0, .04};
11
12     // aim base-class pointer at derived-class object (allowed)
13     SalariedEmployee* salariedPtr{&salariedCommission};
14
15     // invoke base-class member functions on derived-class
16     // object through base-class pointer (allowed)
17     std::string name{salariedPtr->getName()};
18     double salary{salariedPtr->getSalary()};
19
20     // attempt to invoke derived-class-only member functions
21     // on derived-class object through base-class pointer (disallowed)
22     double grossSales{salariedPtr->getGrossSales()};
23     double commissionRate{salariedPtr->getCommissionRate()};
24     salariedPtr->setGrossSales(8000.0);
25 }
```

GNU C++ compiler error messages:

```

fig10_09.cpp: In function ‘int main()’:
fig10_09.cpp:22:35: error: ‘class SalariedEmployee’ has no member named
‘getGrossSales’
22 |     double grossSales{salariedPtr->getGrossSales()};
|           ^~~~~~
```

Fig. 10.9 | Attempting to call derived-class-only functions via a base-class pointer. (Part I of 2.)

```
fig10_09.cpp:23:39: error: 'class SalariedEmployee' has no member named  
'getCommissionRate'  
23 |     double commissionRate{salariedPtr->getCommissionRate();};  
      |                                     ^~~~~~  
fig10_09.cpp:24:17: error: 'class SalariedEmployee' has no member named  
'setGrossSales'  
24 |     salariedPtr->setGrossSales(8000.0);  
      |                                     ^~~~~~
```

Fig. 10.9 | Attempting to call derived-class-only functions via a base-class pointer. (Part 2 of 2.)

Lines 9–10 create a `SalariedCommissionEmployee` object. Line 13 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 17–18 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 22–24 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 member functions of the handle's class type—in this case, those of base-class `SalariedEmployee`.



Checkpoint

1 (*True/False*) With `public` inheritance, an object of a base class can be treated as an object of its derived class.

Answer: False. With `public` inheritance, an object of a derived class can be treated as an object of its base class.

2 (*Fill-in*) The *is-a* relationship applies only from a _____ to its direct and indirect _____.

Answer: derived class, base classes.

3 (*True/False*) The compiler allows us to invoke only base-class member functions via a base-class pointer.

Answer: True.

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 the 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

Suppose the shape classes `Circle`, `Triangle`, `Rectangle` and `Square` 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 shapes, it would be convenient to treat them all generally as base-class `Shape` objects. Using a base-class `Shape` pointer to invoke function `draw`, we could draw any shape. At runtime, the program would dynamically determine 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)—determines which member function to invoke.

10.7.2 Declaring virtual Functions

To enable this runtime polymorphic behavior, 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,

```
virtual void draw() const;
```

would appear in base class `Shape` to indicate that `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 inherits its base class's `virtual` function implementation.



Checkpoint

1 (*True/False*) An overridden function in a derived class must have the same signature as the base-class function it overrides.

Answer: True.

2 (*True/False*) Once a function is declared `virtual`, it must explicitly be declared `virtual` in all classes derived directly or indirectly from that base class.

Answer: False. Actually, once a function is declared `virtual`, it's `virtual` in all classes derived directly or indirectly from that base class.

3 (*Discussion*) Assume that class `Shape` contains the following function prototype:

```
virtual void draw() const;
```

If class `Square` inherits from `Shape` but does not override `draw`, would class `Square` have a `draw` member function? Explain why or why not.

Answer: Yes, class `Square` would have a `draw` member function because it would inherit its base class `Shape`'s `virtual` function implementation.

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`, based on the performance requirements of your applications.

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 via a specific object's *name* 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.



Checkpoint

1 (*Fill-in*) Choosing the appropriate `virtual` function to call at execution time is known as _____ or _____.

Answer: dynamic binding, late binding.

2 (*Fill-in*) When a `virtual` function is called via a specific object's name using the dot operator, an optimizing compiler can resolve at compile-time the function to call. This is called _____.

Answer: static binding.

10.7.4 virtual Functions in the SalariedEmployee Hierarchy

Let's see how `virtual` functions could enable runtime polymorphic behavior in our hierarchy. We made only two modifications to each class's header to enable the behavior you'll see in Fig. 10.10. In class `SalariedEmployee`'s header (Fig. 10.1), we modified the `earnings` and `toString` prototypes (lines 17–18)

```
double earnings() const;
std::string toString() const;
```

to include the `virtual` keyword:

```
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)

```
double earnings() const;
std::string toString() const;
```

to include the `override` keyword (discussed after Fig. 10.10):

```
double earnings() const override;
std::string toString() const override;
```

The base class's `earnings` and `toString` functions are `virtual`, so the derived class `SalariedCommissionEmployee`'s versions *override* `SalariedEmployee`'s versions. There were no changes to `SalariedEmployee`'s or `SalariedCommissionEmployee`'s member-function implementations, so we reused the versions of Figs. 10.2 and 10.5.

Runtime Polymorphic Behavior

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 10–21 create a `SalariedEmployee` and a `SalariedCommissionEmployee`, then use their object names to show their `toString` results. This will help you confirm the dynamic binding results later in the program. Lines 27–38 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 41 aims the base-class pointer `salariedPtr` at the derived-class object `salariedCommission`. Line 48 invokes `toString` via the base-class pointer. As you can see in the output, the derived-class `salariedCommission` object's `toString` 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 object's type.



```

1 // fig10_10.cpp
2 // Introducing polymorphism, virtual functions and dynamic binding.
3 #include <iostream>
4 #include <iomanip>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7
8 int main() {
9     // create base-class object
10    SalariedEmployee salaried{"Sierra Dembo", 500.0};
11
12    // create derived-class object
13    SalariedCommissionEmployee salariedCommission{
14        "Ivano Lal", 300.0, 5000.0, .04};
15
16    // output objects using static binding
17    std::cout << std::format("{}\n{}:{}\n{}\n",
18        "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND",
19        "DERIVED-CLASS OBJECTS WITH STATIC BINDING",
20        salaried.toString(), // static binding
21        salariedCommission.toString()); // static binding
22
23    std::cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND\n"
24        << "DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING\n\n";
25
26    // natural: aim base-class pointer at base-class object
27    SalariedEmployee* salariedPtr{&salaried};

```

Fig. 10.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part I of 2.)

```
28     std::cout << std::format("{}\n{}:\n{}\n",
29         "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
30         "TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
31         salariedPtr->toString()); // base-class version
32
33     // natural: aim derived-class pointer at derived-class object
34     SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
35     std::cout << std::format("{}\n{}:\n{}\n",
36         "CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER",
37         "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
38         salariedCommissionPtr->toString()); // derived-class version
39
40     // aim base-class pointer at derived-class object
41     salariedPtr = &salariedCommission;
42
43     // runtime polymorphism: invokes SalariedCommissionEmployee
44     // via base-class pointer to derived-class object
45     std::cout << std::format("{}\n{}:\n{}\n",
46         "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
47         "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
48         salariedPtr->toString()); // derived-class version
49 }
```

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH STATIC BINDING:

name: Sierra Dembo
salary: \$500.00

name: Ivano Lal
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: Sierra Dembo
salary: \$500.00

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Ivano Lal
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: Ivano Lal
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 31). When `salariedPtr` points to a `SalariedCommissionEmployee` object, class `SalariedCommissionEmployee`'s `toString` function is invoked (line 48). 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

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

`override` Keyword

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



Checkpoint

1 *(Code)* Assume that class `Shape` contains the following function prototype:

```
virtual void draw() const;
```

and that you're defining class `Square`, which inherits from `Shape` and will override `draw`. Write `draw`'s function prototype that you'd place in class `Square`'s definition.

Answer: `void draw() const override;`

2 *(Fill-in)* When the same member function call via a base-class pointer to various objects takes on many forms, this is runtime _____ behavior.

Answer: polymorphic.

3 *(True/False)* To help you prevent errors, apply `override` to the prototype of every derived-class function that overrides a `virtual` base-class function.

11. "C.82: Don't Call Virtual Functions in Constructors and Destructors." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual>.

12. "C.128: Virtual Functions Should Specify Exactly One of `virtual`, `override`, or `final`." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override>.

Answer: True. [Insight: Declaring derived-class 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 and prevents you from accidentally hiding a base-class function with the same name but a different signature.]

10.7.5 virtual Destructors

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

```
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`.



CG



Checkpoint

I (Discussion) Explain why in Modern C++, the `virtual` destructor definition for most classes is written as

```
virtual ~ClassName() = default;
```

Answer: This enables you to declare the destructor `virtual` and still have the compiler generate a default destructor for the class—via the notation `= default`.



SE

10.7.6 final Member Functions and Classes

A `virtual` function that's declared `final` in its prototype, as in

```
returnType someFunction(parameters) final;
```

cannot be overridden in any derived class. In a multilevel class hierarchy, this guarantees that the `final` member-function definition will be used by all subsequent direct *and* indirect derived classes.

You can declare a class `final` to prevent it from being used as a base class, as in

```
class MyClass final {
    // class body
};
```

or

```
class DerivedClass : public BaseClass final {
    // class body
};
```

Attempting to override a `final` member function or inherit from a `final` base class results in a compilation error.



Err

13. “C.35: A Base Class Destructor Should Be Either `public` and `virtual`, or `protected` and Non-`virtual`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-virtual>.



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

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 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}



Checkpoint

1 (*Fill-in*) A `virtual` function that's declared _____ in its prototype cannot be overridden in any subsequent derived class.

Answer: `final`. [Insight: In a multilevel class hierarchy, this guarantees that the `final` member-function definition will be used by all subsequent direct and indirect derived classes.]

2 (*Code*) Write the first line of a `Square` class definition that inherits from class `Shape`. Indicate that class `Square` cannot be a base class.

Answer: `class Square : public Shape final`

3 (*Fill-in*) A benefit of declaring a `virtual` function `final` is that once the compiler knows a `virtual` function cannot be overridden, the compiler might be able to determine at compile-time the correct function to call. This is an optimization called _____.

Answer: devirtualization.

10.8 Abstract Classes and Pure virtual Functions



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. 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 and 10.11.

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 an abstract class is `Shape` in Section 10.2.2's hierarchy. 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 objects.** For example, if someone tells you to "draw the two-dimensional

- 14. Matt Godbolt, "Optimizations in C++ Compilers," November 12, 2019. Accessed April 18, 2023. <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.
- 16. Godbolt, "Optimizations in C++ Compilers."
- 17. Sy Brand, "The Performance Benefits of Final Classes," March 2, 2020. Accessed April 18, 2023. <https://devblogs.microsoft.com/cppblog/the-performance-benefits-of-final-classes/>.

shape,” what shape would you draw? Concrete classes provide the specifics that make it possible to instantiate objects.

10.8.1 Pure virtual Functions

A 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

```
virtual void draw() const = 0; // pure virtual function
```

The “= 0” is a **pure specifier**. Pure virtual functions do not provide implementations. Each concrete derived class must override 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 10.11, an abstract class with all pure virtual functions is sometimes called a **pure abstract class** or an **interface**.

A SE
A pure virtual function is 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 implement a draw function. There’s no way to draw a generic space object without knowing which specific type of space object is being drawn.

A SE
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 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 given type. However, the write call itself is no different from the write to any other device in the system—place some bytes from memory onto that device.

Consider an object-oriented operating system that uses an abstract base class to provide an interface appropriate for all device drivers. 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.

A SE
This architecture also allows new devices to be added to a system easily. The user can just plug in the device and install its 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.



Checkpoint

1 (*Fill-in*) A class that you cannot instantiate is a(n) _____ class and typically defines a common public interface for the classes derived from it in a class hierarchy.

Answer: abstract

2 (*Code*) Explain why a class would include the following function declaration:

```
virtual void print() const = 0;
```

Answer: This would indicate that `print` is a pure `virtual` function and that the class containing it is abstract. This enables us to require that all concrete derived classes of this class should implement `print`.

3 (*True/False*) We cannot instantiate objects of an abstract base class or declare pointers or references of the abstract-base-class type.

Answer: False. Actually, 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.

10.9 Case Study: Payroll System Using Runtime Polymorphism

Let's 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:

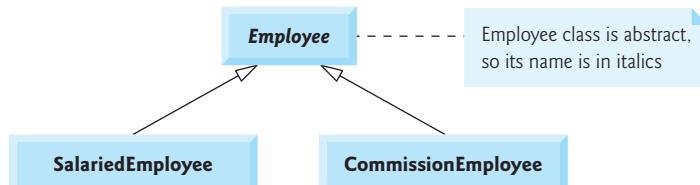
1. *Salaried employees are paid a fixed salary regardless of their hours worked.*
2. *Commission employees are paid a percentage of their sales.*

The company wants to implement a C++ program that performs its payroll calculations polymorphically.



Many hierarchies start with an abstract base class followed by a row of derived classes that are `final`. We use the abstract class `Employee` to represent the general concept of an employee and define the “interface” to the hierarchy—that is, the member functions all `Employee` derived classes must have, so a program can invoke them on *all* `Employee` objects. Also, regardless of how earnings are calculated, each employee has a name. So, we'll define a private data member `m_name` in `Employee`.

`SalariedEmployee` and `CommissionEmployee` are `final` classes that we derive directly from `Employee`. Each is a *leaf node* in the class hierarchy and cannot be a base class. The following UML class diagram shows our runtime polymorphic payroll application’s inheritance hierarchy. The abstract class `Employee` is *italicized*, per the UML’s convention.



A derived class can inherit interface and/or implementation from a base class. Hierarchies designed for **interface inheritance** tend to have concrete definitions of their functionality lower in the hierarchy. A base class declares one or more functions that should be defined by every derived class, but the individual derived classes provide the implementations of the function(s).



The following subsections implement the `Employee` class hierarchy. The first three each define one of the abstract or concrete classes. The last contains 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 generally applies to all `Employees`, but each calculation depends on the kind of `Employee`. 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.

The test program contains a vector of `Employee` pointers. Each points to an object that is *an Employee*—specifically, any object of a concrete derived class of `Employee`. The program iterates through the vector and polymorphically calls each `Employee`'s `earnings` function. **Making `earnings` 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. Each class derived from `Employee` will override `toString` to return the 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 `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 always call 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.

	earnings	toString
<code>Employee</code>	pure virtual	<code>name: m_name</code>
<code>Salaried-Employee</code>	<code>m_salary</code>	<code>name: m_name</code> <code>salary: m_salary</code>
<code>Commission-Employee</code>	<code>m_commissionRate * m_grossSales</code>	<code>name: m_name</code> <code>gross sales: m_grossSales</code> <code>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 it does not provide an implementation. Each derived class overrides this function to provide an appropriate implementation. We do not list base class `Employee`'s `get` and `set` functions because the derived classes do not override them. Each function 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),
- a defaulted virtual destructor (line 10),
- a `set` function to set the name (line 12),
- a `get` function to return 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 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`.

```

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 };

```

Fig. 10.11 | `Employee` abstract base class.

`Employee` Class Member-Function Definitions

Figure 10.12 contains the `Employee` class's member-function definitions. No implementation is provided for `virtual` function `earnings`. The `virtual` function `toString` implementation (lines 17–19) 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).

```

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 <format>
5 #include "Employee.h" // Employee class definition
6
7 // constructor
8 Employee::Employee(std::string_view name) : m_name{name} {} // empty body
9
10 // set name
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // get name
14 std::string Employee::getName() const {return m_name;}
15
16 // return string representation of an Employee
17 std::string Employee::toString() const {
18     return std::format("name: {}", getName());
19 }

```

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),
- a `default virtual` destructor (line 11),
- a `set` function to assign a new non-negative 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.

```

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;

```

Fig. 10.13 | SalariedEmployee class header. (Part I of 2.)

```

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 };

```

Fig. 10.13 | SalariedEmployee class header. (Part 2 of 2.)

SalariedEmployee Class Member-Function Definitions

Figure 10.14 defines SalariedEmployee’s member functions:

- Its constructor passes name to the Employee constructor (line 9) to initialize the inherited private data member that’s not directly accessible in the derived class.
- Function earnings (line 27) 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 be abstract.
- SalariedEmployee’s toString function (lines 30–33) overrides Employee’s toString. If it did not, the class would inherit Employee’s, which returns a string containing the employee’s name. SalariedEmployee’s toString returns a string containing the Employee::toString() result and the SalariedEmployee’s salary.

SalariedEmployee’s header declared member functions earnings and toString as override to ensure that we correctly override them. These are virtual in base class Employee, so they remain virtual throughout the class hierarchy.

```

1 // Fig. 10.14: SalariedEmployee.cpp
2 // SalariedEmployee class member-function definitions.
3 #include <iostream>
4 #include <stdexcept>
5 #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7 // constructor
8 SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9     : Employee{name} {
10     setSalary(salary);
11 }
12
13 // set salary
14 void SalariedEmployee::setSalary(double salary) {
15     if (salary < 0.0) {
16         throw std::invalid_argument("Weekly salary must be >= 0.0");
17     }
18
19     m_salary = salary;
20 }

```

Fig. 10.14 | SalariedEmployee class implementation file. (Part 1 of 2.)

```

21
22 // return salary
23 double SalariedEmployee::getSalary() const {return m_salary;}
24
25 // calculate earnings;
26 // override pure virtual function earnings in Employee
27 double SalariedEmployee::earnings() const {return getSalary();}
28
29 // return a string representation of SalariedEmployee
30 std::string SalariedEmployee::toString() const {
31     return std::format("{}\n{}: ${:.2f}", Employee::toString(),
32                         "salary", getSalary());
33 }
```

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 8–12) that takes a name, gross sales and commission rate, then passes the name to Employee’s constructor (line 9) to initialize the inherited data member,
- *set* functions (lines 15–21 and 27–34) to assign new values to data members `m_grossSales` and `m_commissionRate`,
- *get* functions (lines 24 and 37–39) to 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) to return a `string` containing the `Employee::toString()` result, gross sales and commission rate.

```

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;
```

Fig. 10.15 | CommissionEmployee class header. (Part 1 of 2.)

```

16    void setCommissionRate(double commissionRate);
17    double getCommissionRate() const;
18
19    // keyword override signals intent to override
20    double earnings() const override; // calculate earnings
21    std::string toString() const override; // string representation
22
23 private:
24    double m_grossSales{0.0};
25    double m_commissionRate{0.0};
26 };

```

Fig. 10.15 | CommissionEmployee class header. (Part 2 of 2.)

```

1 // Fig. 10.16: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <iostream>
4 #include <stdexcept>
5 #include "CommissionEmployee.h" // CommissionEmployee class definition
6
7 // constructor
8 CommissionEmployee::CommissionEmployee(std::string_view name,
9     double grossSales, double commissionRate) : Employee{name} {
10    setGrossSales(grossSales);
11    setCommissionRate(commissionRate);
12 }
13
14 // set gross sales amount
15 void CommissionEmployee::setGrossSales(double grossSales) {
16    if (grossSales < 0.0) {
17        throw std::invalid_argument("Gross sales must be >= 0.0");
18    }
19
20    m_grossSales = grossSales;
21 }
22
23 // return gross sales amount
24 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
25
26 // set commission rate
27 void CommissionEmployee::setCommissionRate(double commissionRate) {
28    if (commissionRate <= 0.0 || commissionRate >= 1.0) {
29        throw std::invalid_argument(
30            "Commission rate must be > 0.0 and < 1.0");
31    }
32
33    m_commissionRate = commissionRate;
34 }
35

```

Fig. 10.16 | CommissionEmployee class implementation file. (Part 1 of 2.)

```

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 std::string CommissionEmployee::toString() const {
48     return std::format("{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::toString(),
49                        "gross sales", getGrossSales(),
50                        "commission rate", getCommissionRate());
51 }

```

Fig. 10.16 | CommissionEmployee class implementation file. (Part 2 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 16–17 create objects of each concrete derived class. Lines 20–23 output each employee’s information and earnings. These lines use the variable names for each object, so the compiler can identify each object’s type at compile-time to determine which `toString` and `earnings` functions to call.

```

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 "Employee.h"
7 #include "SalariedEmployee.h"
8 #include "CommissionEmployee.h"
9
10
11 void virtualViaPointer(const Employee* baseClassPtr); // prototype
12 void virtualViaReference(const Employee& baseClassRef); // prototype
13
14 int main() {
15     // create derived-class objects
16     SalariedEmployee salaried{"Pierre Simon", 800.0};
17     CommissionEmployee commission{"Sierra Dembo", 10000, .06};
18

```

Fig. 10.17 | Processing Employee derived-class objects with variable-name handles then polymorphically using base-class pointers and references. (Part 1 of 3.)

```

19 // output each Employee
20 std::cout << "EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES\n"
21     << std::format("{}\n{}{:2f}\n{}\n{}{:2f}\n{}\n",
22                     salaried.toString(), "earned $", salaried.earnings(),
23                     commission.toString(), "earned $", commission.earnings());
24
25 // create and initialize vector of base-class pointers
26 std::vector<Employee*> employees{&salaried, &commission};
27
28 std::cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA"
29     << " DYNAMIC BINDING\n\n";
30
31 // call virtualViaPointer to print each Employee
32 // and earnings using dynamic binding
33 std::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 std::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     std::cout << std::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     std::cout << std::format("{}\nearned ${:.2f}\n{}\n",
59                             baseClassRef.toString(), baseClassRef.earnings());
60 }

```

EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES

name: Pierre Simon
 salary: \$800.00
 earned \$800.00

name: Sierra Dembo
 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 2 of 3.)

```
EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
name: Pierre Simon
salary: $800.00
earned $800.00

name: Sierra Dembo
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: Pierre Simon
salary: $800.00
earned $800.00

name: Sierra Dembo
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 26 creates and initializes the vector `employees`, containing two `Employee` pointers aimed at the objects `salaried` and `commission`. 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`

Lines 35–37 traverse the vector `employees` and invoke `virtualViaPointer` (lines 50–53) with each element as an argument. This function receives in its `baseClassPtr` parameter 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 call correctly invokes the virtual function on the object to which `baseClassPtr` currently points. The output shows that each class's appropriate functions are invoked and display the correct information. Obtaining each `Employee`'s `earnings` via runtime polymorphism produces the same results as lines 22 and 23.



Function `virtualViaReference`

Lines 43–45 traverse `employees` and invoke `virtualViaReference` (lines 57–60) with each element as an argument. This function receives in its `baseClassRef` parameter (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 also occurs with base-class references. Each function invocation calls the `virtual` function on the object `baseClassRef` references at runtime—another example of dynamic binding. The output produced using base-class references is identical to that produced using base-class pointers and via static binding earlier in the program.



Checkpoint

1 (*True/False*) A leaf node in a class hierarchy cannot be a base class.

Answer: True.

2 (*True/False*) Hierarchies designed for interface inheritance tend to have concrete definitions of their functionality higher in the hierarchy.

Answer: False. Actually, with interface inheritance, concrete definitions of the functionality tend to appear lower in the hierarchy.

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. 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 to support polymorphism at execution time. You’ll see that this can be accomplished through three levels of pointers—that is, **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 `Employee`, `SalariedEmployee` and `CommissionEmployee`.

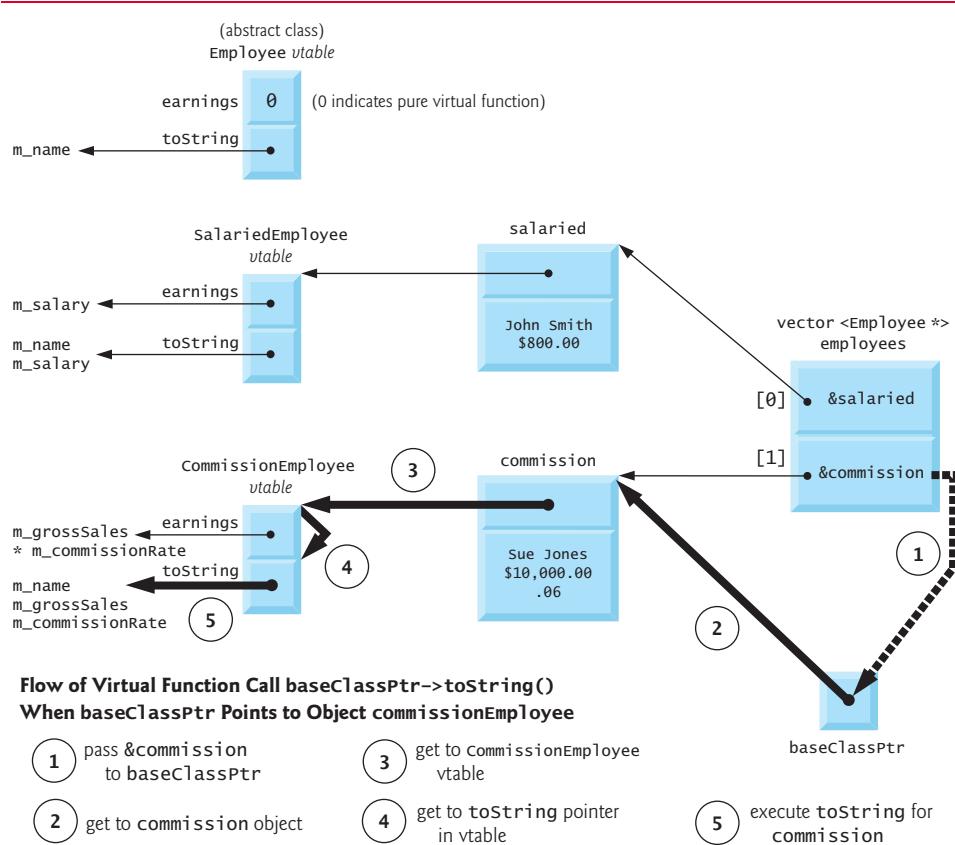


Fig. 10.18 | How virtual function calls work.

Employee Class vtable

The first function pointer in the `Employee` class `vtable` is set to 0 (i.e., `nullptr`) because `earnings` is a pure virtual function with no implementation. The second points to `toString`, which returns a string containing the employee's name. We've abbreviated each `toString` function's output in this figure to conserve space—note the pointers in the `vtables` actually point to each function's code. Any class with one or more pure virtual functions (represented with the value 0) in its `vtable` is an abstract class. `SalariedEmployee` and `CommissionEmployee` are concrete classes—they have no `nullptrs` in their `vtables`.

SalariedEmployee Class vtable

The `earnings` function pointer in the `SalariedEmployee` `vtable` points to that class's override of the `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

Each concrete class in our `Employee` case study provides `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 were to inherit `toString` and not override it, this function's *vtable* pointer would 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 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 each object a pointer to the class's *vtable*. This pointer is usually 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 contains the addresses of the objects on which the `virtual` functions will be called. The *vtable* diagram's rightmost column depicts the vector `employees` containing these `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 the `baseClassPtr` parameter at the `commission` object.
2. Dereference that pointer to access the `commission` object, which begins with a pointer to class `CommissionEmployee`'s *vtable*.

3. Dereference the *vtable* pointer to access class `CommissionEmployee`'s *vtable*.
4. Skip the offset to select the `toString` function pointer.¹⁸
5. Execute `toString` for the `commission` object, returning a `String` containing the employee's name, gross sales and commission rate.

The *vtable* diagram's data structures may appear complex. The compiler hides this complexity from you, making runtime polymorphic programming straightforward. The pointer dereferencing operations and memory accesses for each `virtual` function call require additional execution time. The *vtables* and *vtable* pointers added to the objects require some additional memory.

Runtime polymorphism is efficient, as typically implemented with `virtual` functions and dynamic binding in C++. 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 each object's size.



Checkpoint

1 (*True/False*) The overhead of runtime polymorphism—in terms of additional memory consumption and processor time—is insignificant, so you should feel free to use it in your applications.

Answer: False. Actually, understanding the overhead of runtime polymorphism—in terms of additional memory consumption and processor time—can help you determine when to use runtime polymorphism in your applications and when to avoid it.

2 (*Fill-in*) When C++ compiles a class with one or more `virtual` functions, it builds a _____ for that class. This contains pointers to the class's `virtual` functions.

Answer: virtual function table (*vtable*)

3 (*True/False*) The *vtable* diagram's complexity discourages programmers from doing polymorphic programming.

Answer: False. Actually, although the *vtable* diagram's data structures may appear complex, the compiler hides this complexity from you, making runtime polymorphic programming straightforward.

10.11 Program to an Interface, Not an Implementation¹⁹

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 creates **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.

18. In practice, *Steps 3 and 4* can be implemented as a single machine instruction.

19. Defined in Gamma et al., pp. 17–18; also discussed in Joshua Bloch, *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

SE 

Tight coupling can make modifying class hierarchies difficult. Consider modifying Section 10.9's `Employee` hierarchy to support retirement plans. There are many retirement plan types, including 401(k)s 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. You can see that 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 are 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: 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.²⁰

SE 

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 calculates 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 `Employee` 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**.

CG 

The C++ Core Guidelines recommend inheriting from pure abstract classes rather than classes with implementation details.^{21,22,23,24}

20. 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 textbook *Java How to Program*.

21. "I.25: Prefer Abstract Classes as Interfaces to Class Hierarchies." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abstract>.

22. "C.121: If a Base Class Is Used as an Interface, Make It a Pure Abstract Class." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-abstract>.

23. "C.122: Use Abstract Classes As Interfaces When Complete Separation of Interface and Implementation Is Needed." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-separation>.

24. "C.129: When Designing a Class Hierarchy, Distinguish Between Implementation Inheritance and Interface Inheritance." Accessed April 18, 2023. <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 they're 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.



10.11.1 Rethinking the Employee Hierarchy—CompensationModel Interface

Let's refactor Section 10.9's `Employee` hierarchy with composition and an interface. We can say that each `Employee` *has a* `CompensationModel`. Figure 10.19 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.



```

1 // Fig. 10.19: 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 };

```

Fig. 10.19 | CompensationModel “interface” is a pure abstract base class.

10.11.2 Class Employee

Figure 10.20 defines our new `Employee` class. Each `Employee` *has a* pointer to the implementation of its `CompensationModel` (line 16). This class is declared `final`, so it cannot be used as a base class.

```

1 // Fig. 10.20: 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

```

Fig. 10.20 | An Employee “has a” CompensationModel. (Part 1 of 2.)

```

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 };

```

Fig. 10.20 | An Employee “has a” CompensationModel. (Part 2 of 2.)

Figure 10.21 defines Employee’s member functions:

- The constructor (lines 10–11) 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.²⁵
- The setCompensationModel member function (lines 15–17) enables the client code to change an Employee’s CompensationModel by aiming m_modelPtr at a different CompensationModel object. This technique is known as **property injection**.
- The earnings member function (lines 20–22) 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 25–27) creates an Employee’s string representation. This consists of the Employee’s name, followed by the compensation information we get by calling the CompensationModel implementation’s toString member function via m_modelPtr.



Constructor injection and property injection are both forms of **dependency injection**. You specify part of an object’s behavior by providing a pointer or reference to an object that defines the behavior.²⁶ In this example, a CompensationModel provides the behavior that enables an Employee to calculate earnings and generate a string representation.

```

1 // Fig. 10.21: Employee.cpp
2 // Class Employee member-function definitions.
3 #include <format>
4 #include <string>
5 #include "CompensationModel.h"

```

Fig. 10.21 | Class Employee member-function definitions. (Part 1 of 2.)

25. With dependency injection, the CompensationModel object must exist longer than the Employee object to prevent a runtime logic error known as a dangling pointer. We discuss dangling pointers in Section 11.6.
26. “Dependency Injection.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Dependency_injection.

```

6 #include "Employee.h"
7
8 // constructor performs "constructor injection" to initialize
9 // the CompensationModel pointer to a CompensationModel implementation
10 Employee::Employee(std::string_view name, CompensationModel* modelPtr)
11   : m_name{name}, m_modelPtr{modelPtr} {}
12
13 // set function performs "property injection" to change the
14 // CompensationModel pointer to a new CompensationModel implementation
15 void Employee::setCompensationModel(CompensationModel* modelPtr) {
16   m_modelPtr = modelPtr;
17 }
18
19 // use the CompensationModel to calculate the Employee's earnings
20 double Employee::earnings() const {
21   return m_modelPtr->earnings();
22 }
23
24 // return string representation of Employee object
25 std::string Employee::toString() const {
26   return std::format("{}\n{}", m_name, m_modelPtr->toString());
27 }
```

Fig. 10.21 | Class Employee member-function definitions. (Part 2 of 2.)

10.11.3 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. 10.22–10.23) 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.

```

1 // Fig. 10.22: 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 }
```

Fig. 10.22 | Salaried implements the CompensationModel interface.

```

1 // Fig. 10.23: Salaried.cpp
2 // Salaried compensation model member-function definitions.
3 #include <iostream>
4 #include <stdexcept>
5 #include "Salaried.h" // class definition
6
7 // constructor
8 Salaried::Salaried(double salary) : m_salary{salary} {
9     if (m_salary < 0.0) {
10         throw std::invalid_argument("Weekly salary must be >= 0.0");
11     }
12 }
13
14 // override CompensationModel pure virtual function earnings
15 double Salaried::earnings() const {return m_salary;}
16
17 // override CompensationModel pure virtual function toString
18 std::string Salaried::toString() const {
19     return std::format("salary: ${:.2f}", m_salary);
20 }
```

Fig. 10.23 | Salaried compensation model member-function definitions.

Commission Derived Class of CompensationModel

A Commission compensation model (Figs. 10.24–10.25) defines how to pay an Employee commission based on gross sales. The class contains data members `m_grossSales` and `m_commissionRate` 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.

```

1 // Fig. 10.24: Commission.h
2 // Commission implements the CompensationModel interface.
3 #pragma once
4 #include <iostream>
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 }
```

Fig. 10.24 | Commission implements the CompensationModel interface.

```

1 // Fig. 10.25: Commission.cpp
2 // Commission member-function definitions.
3 #include <format>
4 #include <stdexcept>
5 #include "Commission.h" // class definition
6
7 // constructor
8 Commission::Commission(double grossSales, double commissionRate)
9     : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
10
11     if (m_grossSales < 0.0) {
12         throw std::invalid_argument("Gross sales must be >= 0.0");
13     }
14
15     if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
16         throw std::invalid_argument(
17             "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 std::string Commission::toString() const {
28     return std::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29                         m_grossSales, m_commissionRate);
30 }
```

Fig. 10.25 | Commission member-function definitions.

10.11.4 Testing the New Hierarchy

We've created our `CompensationModel` interface and derived-class implementations defining how `Employee`s get paid. Now, let's create `Employee` objects and initialize each with an appropriate concrete `CompensationModel` implementation (Fig. 10.26).

```

1 // fig10_26.cpp
2 // Processing Employees with various CompensationModels.
3 #include <format>
4 #include <iostream>
5 #include <vector>
6 #include "Employee.h"
7 #include "Salaried.h"
8 #include "Commission.h"
9 
```

Fig. 10.26 | Processing Employees with various CompensationModels. (Part 1 of 2.)

```

10 int main() {
11     // create CompensationModels and Employees
12     Salaried salaried{800.0};
13     Employee salariedEmployee{"Pierre Simon", &salaried};
14
15     Commission commission{10000, .06};
16     Employee commissionEmployee{"Sierra Dembo", &commission};
17
18     // create and initialize vector of Employees
19     std::vector employees{salariedEmployee, commissionEmployee};
20
21     // print each Employee's information and earnings
22     for (const Employee& employee : employees) {
23         std::cout << std::format("{}\nearned: ${:.2f}\n\n",
24             employee.toString(), employee.earnings());
25     }
26 }
```

```

Pierre Simon
salary: $800.00
earned: $800.00

Sierra Dembo
gross sales: $10000.00; commission rate: 0.06
earned: $600.00

```

Fig. 10.26 | Processing Employees with various CompensationModels. (Part 2 of 2.)

Line 12 creates a `Salaried` compensation model object (`salaried`). Line 13 creates the `Employee` `salariedEmployee` and injects its `CompensationModel`, passing a pointer to `salaried` as the second constructor argument. Line 15 creates a `Commission` compensation model object (`commission`). Line 16 creates the `Employee` `commissionEmployee` and injects its `CompensationModel`, passing a pointer to `commission` as the second constructor argument.²⁷ Line 19 creates a `vector` of `Employees` and initializes it with the `salariedEmployee` and `commissionEmployee` objects. Finally, lines 22–25 iterate through the `vector`, displaying each `Employee`'s string representation and earnings.

10.11.5 Dependency Injection Design Benefits

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.

27. In this example, both `CompensationModels` will outlive their corresponding `Employee` objects, so this example does not have the potential for the runtime logic error mentioned in footnote 25.

Flexibility If Employees Are Promoted

The interface-based, composition-and-dependency-injection approach 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 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 401(k)s and IRAs), we could say that every `Employee` has a `RetirementPlan`. First, we'd define the `RetirementPlan` interface 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`.



Checkpoint

1 (*True/False*) Implementation inheritance creates tightly coupled classes in which the base class's data members and member functions are inherited into derived classes.

Answer: True.

2 (*Discussion*) Describe how you would define an interface in C++.

Answer: Define a class containing only pure `virtual` functions. Such a class is called a pure abstract class.

3 (*Fill-in*) With _____, a class contains a pointer to an object that provides a behavior required by objects of the class.

Answer: dependency injection.

10.12 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 initially focused on implementation inheritance, then pointed out that decades of experience with real-world, business-critical and mission-critical systems have 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.

A key goal of this chapter was to familiarize you with inheritance mechanics and runtime polymorphism with `virtual` functions. Now, you'll be able to better appreciate newer polymorphism idioms and techniques we show later in the book that can promote ease of modifiability and better performance.

In Chapter 11, we continue our object-oriented programming presentation with operator overloading, which enables existing operators to work with custom class objects. 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.

Exercises

10.1 (*Inheritance Advantage*) Discuss how inheritance saves time during program development and helps prevent errors.

10.2 (*Programming in the General*) How is it that polymorphism enables you to program "in the general" rather than "in the specific"? Discuss the advantages of programming "in the general."

10.3 (*Inheriting Interface vs. Implementation*) Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

10.4 (*Virtual Functions*) What are `virtual` functions? Describe a circumstance in which `virtual` functions would be appropriate.

10.5 (*Dynamic Binding vs. Static Binding*) Distinguish between static binding and dynamic binding. Explain the use of `virtual` functions and the `vtable` in dynamic binding.

10.6 (*Virtual Functions vs. Pure Virtual Functions*) Distinguish between `virtual` functions and pure `virtual` functions.

10.7 (*Polymorphism and Extensibility*) How does polymorphism promote extensibility?

Inheritance Exercises

10.8 (Student Inheritance Hierarchy) Draw an inheritance hierarchy for university students like the one in Section 10.2.1. Use `Student` as the base class of the hierarchy, then include classes `UndergraduateStudent` and `GraduateStudent` that derive from `Student`. Continue to extend the hierarchy to many levels. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might derive from `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might derive from `GraduateStudent`. [Note: You do not need to write any code for this exercise.]

10.9 (Quadrilateral Inheritance Hierarchy) Draw an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the base class of the hierarchy.

10.10 (Richer Shape Hierarchy) The world of shapes is much richer than the shapes included in the inheritance hierarchy of Section 10.2.2. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have the base-class `Shape` from which class `TwoDimensionalShape` and class `ThreeDimensionalShape` are derived. [Note: You do not need to write any code for this exercise.]

10.11 (Composition as an Alternative to Inheritance) Many inheritance hierarchies can be written with composition instead. Rewrite class `SalariedCommissionEmployee` from Section 10.3.2. Rather than inheriting from class `SalariedEmployee`, use composition. So, a `SalariedCommissionEmployee` should have a `SalariedEmployee` object as a data member. Ensure that class `SalariedCommissionEmployee`'s interface includes the member functions previously inherited from class `SalariedEmployee`—each should call the corresponding member function on the composed `SalariedEmployee` object. After you do this, assess the relative merits of the two approaches for designing classes `SalariedEmployee` and `SalariedCommissionEmployee`. Which approach is more natural? Why?

10.12 (Account Inheritance Hierarchy) Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit money into their accounts and withdraw money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold, whereas checking accounts charge a fee per transaction.

Create an inheritance hierarchy containing base-class `Account`. The derived classes `SavingsAccount` and `CheckingAccount` inherit from `Account`. Class `Account` should include one data member of type `cpp_dec_float_50` (from Boost Multiprecision; Section 4.14) to represent the account balance. The class should provide:

- A constructor—The constructor should receive an initial balance and use it to initialize the data member. Validate the initial balance to ensure it's greater than or equal to `0.0`. If not, the balance should be set to `0.0`, and the constructor should display an error message indicating that the initial balance was invalid.
- Member function `deposit`—This function deposits money into the `Account`, ensuring the deposit amount is positive and, if not, throws an `invalid_argument` exception.

- Member function `withdraw`—This function withdraws money from the `Account`, ensuring the withdrawal amount is positive and does not exceed the `Account`'s balance. If the withdrawal amount is invalid, throw an `invalid_argument` exception with an appropriate message.
- Member function `getBalance`—This function returns the current balance.

Derived-class `SavingsAccount` should inherit the functionality of an `Account` and include a data member of type `cpp_dec_float_50` (from Boost Multiprecision) indicating the interest rate (percentage) assigned to the `Account`. The class should provide:

- A constructor—The constructor should receive the initial balance and an initial value for the interest rate.
- Member function `calculateInterest`—This function returns an object of type `cpp_dec_float_50` representing the amount of interest earned by an account. The interest amount is the interest rate (such as 0.05 for 5%) times the account balance.

`SavingsAccount` should inherit member functions `deposit` and `withdrawal` as is without redefining them.

Derived-class `CheckingAccount` should inherit from base-class `Account` and include an additional data member of type `cpp_dec_float_50` (from Boost Multiprecision) that represents the fee charged per transaction. The class should provide:

- A constructor—The constructor receives the initial balance and the fee amount per transaction.
- Redefined member functions `deposit` and `withdraw`—These functions subtract the fee from the account balance whenever either transaction is performed successfully. Functions `deposit` and `withdraw` should invoke their base-class `Account` versions to update the account balance. The `withdraw` function should charge a fee only if money is actually withdrawn (i.e., the amount does not exceed the account balance).

Hint: Define `Account`'s `withdraw` function to return a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.

After defining the `Account` hierarchy, write a program that creates one object of each class and tests its member functions.

Polymorphism Exercises

10.13 (Payroll-System Modification) Modify the payroll system of Section 10.9 to include `private` data member `m_birthDate` in class `Employee`. Use class `Date` from Figs. 9.25–9.26 to represent an employee's birthday. Assume that payroll is processed once per month. Create a vector of `Employee` pointers to store the various employee objects. In a loop, calculate the payroll for each `Employee` polymorphically, and add a \$100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

10.14 (Polymorphic Banking Program Using Account Hierarchy) Develop a polymorphic banking program using the `Account` hierarchy created in Exercise 10.12. Modify the `deposit` and `withdraw` functions in class `Account` to be `virtual` functions to enable poly-

morphic processing. Create a vector of `Account` pointers to `SavingsAccount` and `CheckingAccount` objects. For each `Account` in the vector, allow the user to specify an amount of money to withdraw and an amount of money to deposit. After processing an `Account`, print the updated account balance obtained by invoking the base-class member function `getBalance`.

10.15 (Payroll-System Modification) Modify the payroll system of Section 10.9 to include an additional `Employee` subclass named `HourlyWorker`, representing an employee whose pay is based on an hourly wage and the number of hours worked. Hourly workers receive overtime pay (1.5 times the hourly wage) for all hours over 40 hours worked.

Class `HourlyWorker` should contain private data members `m_wage` (to store the employee's wage per hour) and `m_hours` (to store the hours worked). Provide a concrete implementation of member function `earnings` that calculates the employee's earnings by multiplying the hours worked by the wage per hour. If the number of hours worked is over 40, pay the `HourlyWorker` for the overtime hours. Add a pointer to an `HourlyWorker` object into the vector of `Employee` pointers in `main` and test that the new object works correctly as main polymorphically displays each `Employee`'s string representation and `earnings`.

Programming to an Interface Exercises

10.16 (Programming to an Interface Modification) Modify the payroll system of Section 10.11 to include the additional `CompensationModel` subclass `Hourly`, which determines an employee's pay based on an hourly wage and the number of hours worked, with 1.5 times the hourly wage for all hours over 40 hours worked. Create an `Employee` object that gets paid \$20 per hour and worked 45 hours. Add the new `Employee` object to the payroll system's vector of `Employees` in `main` to test that your new subclass of `CompensationModel` works correctly.

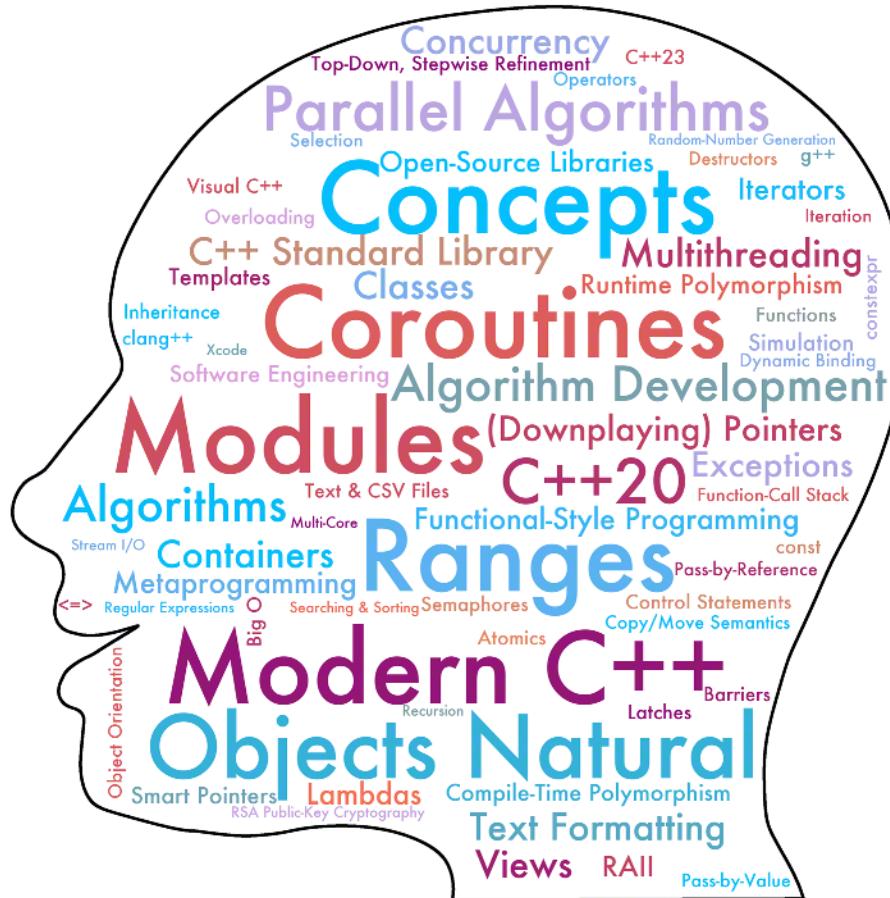
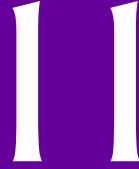
10.17 (Project: Refactoring a Class Hierarchy to Use Interface Inheritance and Dependency Injection) Refactor the `Account` inheritance hierarchy of Exercises 10.12 and 10.14 to use the interface inheritance and dependency injection techniques from Section 10.11. For this exercise, class `Account` should be a `final` class with two data members:

- a `cpp_dec_float_50` (from Boost Multiprecision) for the balance and
- a pointer to an object that implements the `AccountModel` interface.

The `AccountModel` interface should have derived classes `Savings` and `Checking` to implement the deposit and withdrawal specifics for savings and checking accounts, respectively. Test your refactored code with one `Account` that uses the `Savings` model and one that uses the `Checking` model.

This page intentionally left blank

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 when to implement the special member functions for custom types.
 - Understand when objects should be moved vs. copied.
 - Use *rvalue* references and move semantics to eliminate unnecessary copying when objects go out of scope, improving performance.
 - Manage dynamic memory automatically with smart pointers.
 - Craft a **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 **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 <code>string</code>	
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 <code>new</code> and <code>delete</code>	
11.5	Modern C++ Dynamic Memory Management: RAII and Smart Pointers	
11.5.1	Smart Pointers	
11.5.2	Demonstrating <code>unique_ptr</code>	
11.5.3	<code>unique_ptr</code> Ownership	
11.5.4	<code>unique_ptr</code> 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 <code>MyArray</code>	
11.6.3	<code>MyArray</code> Class Definition	
11.6.4	Constructor That Specifies a <code>MyArray</code> 's Size	
11.6.5	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	<code>toString</code> and <code>size</code> Functions	
11.6.10	Overloading the Equality (<code>==</code>) and Inequality (<code>!=</code>) Operators	
11.6.11	Overloading the Subscript (<code>[]</code>) Operator	
11.6.12	Overloading the Unary <code>bool</code> Conversion Operator	
11.6.13	Overloading the Preincrement Operator	
11.6.14	Overloading the Postincrement Operator	
11.6.15	Overloading the Addition Assignment Operator (<code>+=</code>)	
11.6.16	Overloading the Binary Stream Extraction (<code>>></code>) and Stream Insertion (<code><<</code>) Operators	
11.6.17	<code>friend</code> Function <code>swap</code>	
11.7	C++20 Three-Way Comparison Operator (<code><=></code>)	
11.8	Converting Between Types	
11.9	<code>explicit</code> Constructors and Conversion Operators	
11.10	Overloading the Function Call Operator <code>()</code>	
11.11	Wrap-Up Exercises	

11.1 Introduction

This chapter presents **operator overloading**, which enables C++'s existing operators to work with custom class objects. One example of an overloaded operator in standard C++ is `<<`, which we use as

- the stream insertion operator and
- the bitwise left-shift operator (see the Bit Manipulation appendix at <https://deitel.com/cpphtp11>).

Similarly, we use `>>` as

- the stream extraction operator and
- the bitwise right-shift operator (see the Bit Manipulation appendix at <https://deitel.com/cpphtp11>).

You've already used many overloaded operators. Some are built into the core C++ language itself. For example, the `+` operator performs differently, based on its context in integer, floating-point and pointer arithmetic with data of fundamental types. Class `string` also overloads `+` to concatenate `strings`.

You can overload most operators for use with custom 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 and more convenient.

string Class Overloaded Operators Demonstration

In Section 2.8'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 other `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 the 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, when coupled with the **RAII (Resource Acquisition Is Initialization)** strategy, smart pointers 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 considered 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` has elements.

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 (`<=>`)

We introduce C++20’s new three-way comparison operator (`<=>`)—known as the “spaceship operator.”¹

Conversion Operators

We discuss overloaded conversion operators and conversion constructors in more depth, demonstrating how implicit conversions can cause subtle problems. Then, we use `explicit` to prevent those problems.



Checkpoint

1 *(Fill-in)* Forgetting to release dynamically allocated memory that’s no longer needed can cause a(n) _____.

Answer: memory leak.

2 *(Fill-in)* Smart pointers, coupled with _____, can eliminate subtle memory leak issues.

Answer: RAII (Resource Acquisition Is Initialization)

3 *(Fill-in)* The _____ you can define in each class include the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor.

Answer: special member functions

11.2 Using the Overloaded Operators of Standard Library Class `string`

Chapter 8 presented the `string` class. Figure 11.1 demonstrates many of `string`’s overloaded operators and other useful member functions, including `empty`, `substr` and `at`:

- `empty` determines whether a `string` is empty,
- `substr` (for “substring”) returns 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.

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/W55xftItpl4/m/jcIttrMq8agJ?pli=1>.

```
1 // fig11_01.cpp
2 // Standard library string class test program.
3 #include <format>
4 #include <iostream>
5 #include <string>
6 #include <string_view>
7
8 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 9–12 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 15–16 output these three objects, using `cout` and operator `<<`, which `string` and `string_view` overload for output purposes.

```
9     std::string s1{"happy"}; // initialize string from char*
10    std::string s2{" birthday"}; // initialize string from char*
11    std::string s3; // creates an empty string
12    std::string_view v{"hello"}; // initialize string_view from char*
13
14    // output strings and string_view
15    std::cout << "s1: \"\" " << s1 << "\""; s2: \"\" " << s2
16        << "\""; s3: \"\" " << s3 << "\""; v: \"\" " << v << "\n\n";
```

```
s1: "happy"; s2: " birthday"; s3: ""; v: "hello"
```

Comparing `string` Objects with the Equality and Relational Operators

Lines 19–25 show the results of comparing `s2` to `s1` using `string`'s overloaded equality and relational operators. These perform lexicographical comparisons using the numerical values of the characters in each `string`. When you use the `format` function to convert a `bool` to a `string`, it produces "true" or "false".

```
18    // test overloaded equality and relational operators
19    std::cout << "The results of comparing s2 and s1:\n"
20        << std::format("s2 == s1: {} \n", s2 == s1)
21        << std::format("s2 != s1: {} \n", s2 != s1)
22        << std::format("s2 > s1: {} \n", s2 > s1)
23        << std::format("s2 < s1: {} \n", s2 < s1)
24        << std::format("s2 >= s1: {} \n", s2 >= s1)
25        << std::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 30 uses `string` member function `empty`, which returns `true` if the `string` is empty; otherwise, it returns `false`. The object `s3` is empty, as we initialized it with the default constructor.

```
27 // test string member function empty
28 std::cout << "Testing s3.empty():\n";
29
30 if (s3.empty()) {
31     std::cout << "s3 is empty; assigning s1 to s3;\n";
32     s3 = s1; // assign s1 to s3
33     std::cout << std::format("s3 is \"{}\"\n\n", s3);
34 }
35
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

string Copy Assignment Operator

Line 32 demonstrates `string`'s **overloaded copy assignment operator** by assigning `s1` to `s3`. Line 33 outputs `s3` to demonstrate that the assignment worked correctly.

string Concatenation and string-Object Literals

Line 37 demonstrates `string`'s overloaded `+=` operator for **string concatenation assignment**. We append the contents of `s2` to `s1`, modifying its value. Then line 38 outputs the updated `s1`. Line 41 demonstrates that you also may append a C-string literal to a `string` object using `+=`. Line 42 displays the result. Similarly, line 46 concatenates `s1` with a **string-object literal**, indicated by placing the letter `s` immediately after the closing `"` of a `string` literal, as in

`", have a great day!"s`

The preceding literal actually calls a C++ standard library function that returns a `string` object containing the literal's characters. Lines 47–48 display `s1`'s new value.

```
36 // test overloaded string concatenation assignment operator
37 s1 += s2; // test overloaded concatenation
38 std::cout << std::format("s1 += s2 yields s1 = {}\n\n", s1);
39
```

(continued...)

```
40 // test string concatenation with a C string
41 s1 += " to you";
42 std::cout << std::format("s1 += \" to you\" yields s1 = {}\n\n", s1);
43
44 // test string concatenation with a string-object literal
45 using namespace std::string_literals;
46 s1 += ", have a great day!"s; // s after " for string-object Literal
47 std::cout << std::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!
```

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. Line 53 obtains a 14-character substring of `s1` starting at position 0. Line 58 obtains a substring starting from position 15 of `s1`. If you do not provide a second argument to `substr`, it returns the remainder of the `string`.

```
50 // test string member function substr
51 std::cout << std::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 std::cout << std::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!
```

string Copy Constructor

Line 61 creates `string` object `s4`, initializing it with a copy of `s1`. This calls `string`'s **copy constructor**, which copies the contents of `s1` into the new object `s4`. You'll see how to define a copy constructor for a custom class in Section 11.6.

```
60 // test copy constructor
61 std::string s4{s1};
62 std::cout << std::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 `string`'s overloaded copy assignment (`=`) operator to demonstrate that it performs **self-assignment** properly, so `s4` still has the same value after the self-assignment. When we build class `MyArray` later in the chapter, we'll show how to properly implement self-assignment for objects that manage their own memory.

```
64 // test overloaded copy assignment (=) operator with self-assignment
65 std::cout << "assigning s4 to s4\n";
66 s4 = s4;
67 std::cout << std::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 `string`'s constructor that receives a `string_view`, in this case, copying the character data represented by the `string_view` `v` (line 12) into the new `string` `s5`.

```
69 // test string's string_view constructor
70 std::cout << "initializing s5 with string_view v\n";
71 std::string s5{v};
72 std::cout << std::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:

- When using `[]` on a `non-const string`, the function returns a modifiable *lvalue*, which you can place on the left of an assignment (`=`) operator to assign a new value to that location in the `string`, as in lines 77–78.
- When using `[]` on a `const string`, the function returns a nonmodifiable *lvalue* that you can use 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.

```
74 // test using overloaded subscript operator to create lvalue
75 s1[0] = 'H';
76 s1[6] = 'B';
77 std::cout << std::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!
```

string's at Member Function

Class `string`'s **at** member function throws an exception if its argument is an invalid out-of-bounds index. If the index is valid, the `at` function 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. Here, we show the error message produced by GNU C++.

```
80 // test index out of range with string member function "at"
81 try {
82     std::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 std::out_of_range& ex) {
86     std::cout << std::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)
```



Checkpoint

- 1 (*True/False*) `string`'s overloaded `[]` operator performs bounds checking.

Answer: False. Actually, the overloaded `[]` operator does not perform bounds checking, so you must ensure that the index you specify in `[]` is within the `string`'s bounds.

- 2 (*Code*) Write a statement that obtains a 17-character substring of `myString` starting at position 0.

Answer: `myString.substr(0, 17);`

- 3 (*Code*) Write a statement that returns `myString`'s character at index 10. Use a `string` member function that throws a `std::out_of_range` exception if the index is out of bounds.

Answer: `myString.at(10);`

11.3 Operator Overloading Fundamentals

As you saw in Fig. 11.1, `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 you to overload most existing operators 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. You overload an operator 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 you are overloading. For example, the function name `operator+` would overload the addition operator (+). When you overload operators as member functions, they must be **non-static**. You call them on an object of the class and operate on that object.

11.3.2 Operators That Cannot Be Overloaded

The following operators cannot be overloaded:² the . (dot) member-selection operator, the .* pointer-to-member operator³, the :: scope-resolution operator and the ?: conditional operator—though this might be overloadable in the future.⁴ See <https://en.cppreference.com/w/cpp/language/operators> for the complete list of overloadable operators.

11.3.3 Operators That You Do Not Have to Overload

Three operators work with objects of each new class by default:



- For most classes, the **assignment operator** (=) can perform **memberwise assignments** of the data members. The default = operator assigns each data member from the “source” object (on the right) to the “target” object (on the left). As you'll see in 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 **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.⁵

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 April 18, 2023.

3. The .* operator is beyond this book's scope.

4. Matthias Kretz, “Making Operator ?: Overloadable,” October 7, 2019. Accessed April 18, 2023. <https://wg21.link/p0917>.

5. “Operator Overloading.” Accessed April 18, 2023. <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 change the order of evaluation of overloaded operators in an expression.
- An operator's grouping (i.e., associativity) 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 + 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.
- 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 or non-member functions.

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.



Checkpoint

1 (Fill-in) An overloaded operator function's name is the keyword _____ followed by the operator's symbol.

Answer: operator.

2 (True/False) Operators overloaded as member functions must be declared static.

Answer: False. Actually, they must be non-static because they are called on and operate on an object of the class.

3 (Discussion) The assignment operator (=) may be used with most classes to perform memberwise assignments 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). When can such memberwise assignments be dangerous?

Answer: This can be dangerous for classes that have pointer members.

11.4 (Downplaying) Dynamic Memory Management with new and delete

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.^{6,7} 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.

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:

```
Time* timePtr{new Time{}};
```

The **new** operator allocates memory for a **Time** object, calls a default constructor to initialize it and returns a **Time***—a pointer of the type specified to the right of the **new** operator. The preceding statement calls **Time**’s default constructor because we did not supply constructor arguments. 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 **delete**:

```
delete timePtr;
```

This 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. Sometimes the problem might be even worse. If the

-
6. C++ Core Guidelines, “R.11: Avoid Calling **new** and **delete** Explicitly.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-newdelete>.
 7. Operators **new** and **delete** can be overloaded. If you overload **new**, 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 placement **new** and **delete** operators, see https://en.wikipedia.org/wiki/Placement_syntax.

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` dynamically allocated memory, be sure not to `delete` the same memory again, as this 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.  Err  Err

Initializing Dynamically Allocated Objects

You can initialize a newly allocated object with constructor arguments, as in

```
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 built-in array of `ints`:

```
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 initializes 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:

https://en.cppreference.com/w/cpp/language/value_initialization

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* non-negative integral expression.

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.

Using `delete` instead of `delete[]` for an array allocated with `new[]` results in undefined behavior. Some compilers call the destructor only for the first object in the array. To ensure that every object in the array receives a destructor call, always use `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  Err

If there is only one pointer to a block of dynamically allocated memory and the pointer goes out of scope, or if you assign it `nullptr` or a different memory address, a **memory leak occurs**.  Err

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 a 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`.



Checkpoint

1 *(Fill-in)* After you `delete` dynamically allocated memory, be sure not to `delete` the same memory again, as this typically causes a program to crash. One way to guard against this is to immediately set the pointer to _____—deleting such a pointer has no effect.
Answer: `nullptr`.

2 *(Discussion)* You cannot use the range-based `for` statement to iterate over a dynamically allocated array because the compiler must know the number of elements at compile-time. A dynamically allocated array's size is determined at execution time. Describe a workaround to fix this problem.

Answer: Create a C++20 `span` representing the dynamically allocated array and its number of elements, then use range-based `for` to iterate over the `span`.

3 *(Code)* Write a statement that dynamically allocates a `std::string` object, initializes it by calling the default constructor and returns a pointer to the new object.

Answer: `std::string* stringPtr{new std::string{}}`;

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**.^{8,9} The concept is

8. C++ Core Guidelines, “R: Resource Management.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource>.
 9. C++ Core Guidelines, “R.1: Manage Resources Automatically Using Resource Handles and RAII (Resource Acquisition Is Initialization).” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-raii>.

straightforward. For any resource that must be returned to the system when the program is done using it, the program should

- create the object as a local variable in a function—the object’s constructor should acquire the resource while initializing the object,
- use that object as necessary in your program, then
- when the function call terminates, the object goes out of scope—the object’s destructor should release the resource.

11.5.1 Smart Pointers

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 that 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[]` 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 20, Other Topics and a Look Toward the Future of C++.

11.5.2 Demonstrating `unique_ptr`

Figure 11.2 demonstrates a `unique_ptr` pointing to a dynamically allocated object of 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 7. To initialize the `unique_ptr`, line 29 calls the `make_unique` function template, which uses `new` to allocate dynamic memory, then returns a `unique_ptr` to that memory. 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.

```
1 // fig11_02.cpp
2 // Demonstrating unique_ptr.
3 #include <format>
4 #include <iostream>
5 #include <memory>
6
7 class Integer {
8 public:
9     // constructor
10    Integer(int i) : value{i} {
11        std::cout << std::format("Constructor for Integer {}\n", value);
12    }
13 }
```

Fig. 11.2 | Demonstrating `unique_ptr`. (Part 1 of 2.)

```

13
14    // destructor
15    ~Integer() {
16        std::cout << std::format("Destructor for Integer {}\n", value);
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     std::cout << "Creating a unique_ptr that points to an Integer\n";
27
28     // create a unique_ptr object and "aim" it at a new Integer object
29     auto ptr{std::make_unique<Integer>(7)};
30
31     // use unique_ptr to call an Integer member function
32     std::cout << std::format("Integer value: {}\n\nMain ends\n",
33                             ptr->getValue());
34 }
```

```

Creating a unique_ptr that points to an Integer
Constructor for Integer 7
Integer value: 7

Main ends
Destructor for Integer 7
```

Fig. 11.2 | Demonstrating `unique_ptr`. (Part 2 of 2.)

Line 33 uses the `unique_ptr` class's overloaded `->` operator to invoke `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 `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 A Most importantly, using `unique_ptr` 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, 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

Only one `unique_ptr` can own a dynamically allocated object, so assigning one `unique_ptr` to another transfers ownership to the target `unique_ptr`. 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.



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

```
auto ptr{std::make_unique<int[]>(10)};
```

`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[]>`) from 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`:

```
std::cout << ptr[2] << "\n";
```



Checkpoint

1 *(Discussion)* Describe the strategy called RAII—Resource Acquisition Is Initialization for managing dynamically allocated resources like memory.

Answer: For any resource that must be returned to the system when the program is done using it, create the object as a local variable in a function—the object's constructor should acquire the resource during object initialization. Next, use the object in your program. When the function call terminates, the object goes out of scope. At this point, the object's destructor should release the resource.

2 *(Code)* Create a `unique_ptr` object and aim it at a new `std::string` object initialized with "hello":

Answer: `auto ptr{std::make_unique<std::string>("hello")};`

3 *(True/False)* Only one `unique_ptr` can own a dynamically allocated object, so assigning one `unique_ptr` to another is a compilation error.

Answer: False. Actually, assigning one `unique_ptr` to another transfers ownership to the target `unique_ptr`.

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 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 $n - 1$. Alternative 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.¹⁰
- 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.
- 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.¹¹

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 relational operators.
- `MyArrays` know their own size, making it easier to pass them to functions.

-
10. In Chapter 13, Data Structures: 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`.
 11. 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.

- 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 RAI (Resource Acquisition Is Initialization) throughout this example to manage the dynamically allocated memory resources. The class's constructors will dynamically allocate memory as `MyArray` objects are initialized. The class's destructor will deallocate the memory when the 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.



Checkpoint

I *(True/False)* The contents of two pointer-based arrays can be meaningfully compared with equality or relational operators using the array names in an expression like

`a1 == a2`

Answer: False. Actually, they cannot. Array names are simply pointers to where the arrays begin in memory, so the preceding expression compares memory addresses of the arrays' first elements.

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.



Checkpoint

- 1 (*True/False*) The copy constructor and copy assignment operator implement a class's copy semantics.

Answer: True.

- 2 (*Discussion*) What is the key advantage of using move semantics?

Answer: The move constructor and move assignment operator help eliminate costly unnecessary copies of objects that are about to be destroyed.

11.6.2 Using Class MyArray

The program of Figs. 11.3–11.5 demonstrates the `MyArray` class 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 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 its 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*.

```

1 // fig11_03.cpp
2 // MyArray class test program.
3 #include <format>
4 #include <iostream>
5 #include <stdexcept>
6 #include <utility> // for std::move
7 #include "MyArray.h"
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

```

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 ten 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`.

```
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     std::cout << std::format("\nints1 size: {}\ncontents: ", ints1.size())
21         << ints1; // uses overloaded <<
22
23     // print ints2 size and contents
24     std::cout << std::format("\nints2 size: {}\ncontents: ", ints2.size())
25         << 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}
```

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** (`<<`).

```
27     // input and print ints1 and ints2
28     std::cout << "\n\nEnter 17 integers: ";
29     std::cin >> ints1 >> ints2; // uses overloaded >>
30
31     std::cout << "\nints1: " << ints1 << "\nints2: " << 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}
```

Using the Overloaded Inequality Operator (`!=`)

Line 36 tests `MyArray`'s overloaded inequality operator (`!=`) by evaluating the condition

```
ints1 != ints2
```

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 autogenerates `!=` if you provide an `==` operator for your type. Operator `!=` simply returns the opposite of `==`.

```
33 // use overloaded inequality (!=) operator
34 std::cout << "\n\nEvaluating: ints1 != ints2\n";
35
36 if (ints1 != ints2) {
37     std::cout << "ints1 and ints2 are not equal\n\n";
38 }
39
```

```
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.

Lines 44–45 display `ints3`'s size and contents to confirm that `ints3`'s elements were set correctly by the **copy constructor**.

```
40 // create MyArray ints3 by copying ints1
41 MyArray ints3{ints1}; // invokes copy constructor
42
43 // print ints3 size and contents
44 std::cout << std::format("\nints3 size: {}\ncontents: ", ints3.size())
45     << ints3;
46
```

```
MyArray copy constructor

ints3 size: 7
contents: {1, 2, 3, 4, 5, 6, 7}
```



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 **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 (=)

Line 49 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 assigning to an array's name causes a compilation error. Line 51 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 49 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.

Err

SE

```
47 // use overloaded copy assignment (=) operator
48 std::cout << "\n\nAssigning ints2 to ints1:\n";
49 ints1 = ints2; // note target MyArray is smaller
50
51 std::cout << "nints1: " << ints1 << "nints2: " << ints2;
52
```

```
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 56 compares `ints1` and `ints2` with the **overloaded equality operator** (`==`) to confirm they are indeed identical after the assignment in line 49.

```
53 // use overloaded equality (==) operator
54 std::cout << "\n\nEvaluating: ints1 == ints2\n";
55
56 if (ints1 == ints2) {
57     std::cout << "ints1 and ints2 are equal\n\n";
58 }
59
```

```
Evaluating: ints1 == ints2
ints1 and ints2 are equal
```

Using the Overloaded Subscript Operator ([])

Line 61 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 65 uses `ints1[5]` on an assignment’s left side as a modifiable *lvalue*¹² 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 71 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 73–75 catch the exception and display its error message by calling the exception’s `what` member function.

```

60    // use overloaded subscript operator to create an rvalue
61    std::cout << std::format("ints1[5] is {}\n\n", ints1[5]);
62
63    // use overloaded subscript operator to create an lvalue
64    std::cout << "Assigning 1000 to ints1[5]\n";
65    ints1[5] = 1000;
66    std::cout << "ints1: " << ints1;
67
68    // attempt to use out-of-range subscript
69    try {
70        std::cout << "\n\nAttempt to assign 1000 to ints1[15]\n";
71        ints1[15] = 1000; // ERROR: subscript out of range
72    }
73    catch (const std::out_of_range& ex) {
74        std::cout << std::format("An exception occurred: {}\n", ex.what());
75    }
76

```

```

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 []** can be used with other kinds of objects—for example, to select elements from **strings** (collections of characters) and **maps** (collections of key–value pairs, which we’ll discuss in Chapter 13, Data Structures: 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 80 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, lines 82–83 display the new **MyArray**’s size and contents.

12. Recall that an *lvalue* can be declared `const`, in which case it would not be modifiable.

```

77 // initialize ints4 with contents of the MyArray returned by
78 // getArrayByValue; print size and contents
79 std::cout << "\nInitialize ints4 with temporary MyArray object\n";
80 MyArray ints4{getArrayByValue()};
81
82 std::cout << std::format("\nints4 size: {}\ncontents: ", ints4.size())
83     << ints4;
84

```

```

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

`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.

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. 

Creating MyArray ints5 and Initializing It With the `rvalue` Returned By Function `std::move`

The **copy constructor** is called when you initialize one `MyArray` with another that's represented by an *lvalue*. A copy constructor copies its argument's contents. This is similar to a text editor's **copy-and-paste**—after the operation, you have two copies of the data.

C++ also supports **move semantics**,¹³ 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 an ***rvalue reference***, which you'll see is declared with `TypeName&&`. *Rvalue* references may refer only to *rvalues*. Typically, these are temporary objects or objects about to be destroyed—called **expiring values** or **xvalues**. 

13. Klaus Iglberger, “Back to Basics: Move Semantics,” YouTube Video, June 16, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=St0MNEU5b0o>.

Line 88 uses class `MyArray`'s **move constructor** to initialize `MyArray ints5`, then lines 90–91 display 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 standard library function `std::move` (from header `<utility>`). This function casts its argument to an *rvalue* reference,¹⁴ telling the compiler that `ints4`'s contents are no longer needed. So, line 88 forces `MyArray`'s **move constructor** to be called and `ints4`'s contents are *moved* into `ints5`.

```

85 // convert ints4 to an rvalue reference with std::move and
86 // use the result to initialize MyArray ints5
87 std::cout << "\n\nInitialize ints5 with result of std::move(ints4)\n";
88 MyArray ints5{std::move(ints4)}; // invokes move constructor
89
90 std::cout << std::format("\nints5 size: {}\ncontents: ", ints5.size())
91 << ints5
92 << std::format("\n\nSize of ints4 is now: {}", ints4.size());
93

```

Initialize ints5 with result of std::move(ints4)
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 92 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

Line 96 uses class `MyArray`'s **move assignment operator** to move `ints5`'s contents (10, 20 and 30) back into `ints4`, then lines 98–99 display the size and contents of `ints4`. Line 96 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, line 100 outputs `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.

14. More specifically, this is called *xvalue* (for expiring value).

```
94 // move contents of ints5 into ints4
95 std::cout << "\n\nMove ints5 into ints4 via move assignment\n";
96 ints4 = std::move(ints5); // invokes move assignment
97
98 std::cout << std::format("\nints4 size: {}\ncontents: ", ints4.size())
99     << ints4
100    << std::format("\n\nSize of ints5 is now: {}", ints5.size());
101
```

```
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 103 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.

```
102 // check if ints5 is empty by contextually converting it to a bool
103 if (ints5) {
104     std::cout << "\n\nints5 contains elements\n";
105 }
106 else {
107     std::cout << "\n\nints5 is empty\n";
108 }
109
```

```
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 that 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 is required:

```
numbers += 1 # Python does not have a ++ operator
```

We've added a similar capability to the `MyArray` class. Line 111 displays `ints4`'s current contents, then line 112 uses `++ints4` to preincrement the `MyArray`, adding one to each element. This expression's result is the updated `MyArray`. We then use `MyArray`'s overloaded stream insertion operator (`<<`) to display the contents.

```
110 // add one to every element of ints4 using preincrement
111 std::cout << "\nints4: " << ints4;
112 std::cout << "\npreincrementing ints4: " << ++ints4;
113
```

```
ints4: {10, 20, 30}
preincrementing ints4: {11, 21, 31}
```

Postincrementing Every `ints4` Element with the Overloaded `++` Operator

Line 115 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 postincrement operator's implementation, which we'll discuss in Section 11.6.14, produces the outputs showing that `MyArray`'s **copy constructor** and **destructor** were called.

```
114 // add one to every element of ints4 using postincrement
115 std::cout << "\n\npostincrementing ints4: " << ints4++ << "\n";
116 std::cout << "\nints4 now contains: " << ints4;
117
```

```
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 broadcasting overloaded addition assignment operator (`+=`) for adding an `int` value to every `MyArray` element. Line 119 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.

```
118 // add a value to every element of ints4 using +=
119 std::cout << "\n\nAdd 7 to every ints4 element: " << (ints4 += 7)
120     << "\n";
121 }
```

```
Add 7 to every ints4 element: {19, 29, 39}
```

Destroying the `MyArray` Objects That Remain

When `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
```



Checkpoint

1 (*True/False*) When a class contains both a copy constructor and a move constructor, the compiler chooses the move constructor for performance reasons.

Answer: False. Actually, when a class contains both a copy constructor and a move constructor, the compiler chooses the correct one to use based on the context.

2 (*Fill-in*) A(n) _____ reference typically refers to a temporary object or object about to be destroyed. Such objects are called expiring values or _____.

Answer: *rvalue, xvalues*.

3 (*Fill-in*) 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 performs the performance optimization called _____.

Answer: named return value optimization (NRVO).

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.
- `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, automatically deleting 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`.

```
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);
```

Fig. 11.4 | `MyArray` class definition with overloaded operators. (Part 1 of 2.)

```
11 // used by copy assignment operator to implement copy-and-swap idiom
12 friend void swap(MyArray& a, MyArray& b) noexcept;
13
14 public:
15     explicit MyArray(size_t size); // construct a MyArray of size elements
16
17     // construct a MyArray with a braced-initializer list of ints
18     explicit MyArray(std::initializer_list<int> list);
19
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);
```

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

```
explicit MyArray(size_t size); // construct a MyArray of size elements
```

declares a **constructor** that specifies the number of **MyArray** elements. The constructor's definition (Fig. 11.5, lines 15–19) performs several tasks:

- Line 16 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` **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.

```
1 // Fig. 11.5: MyArray.cpp
2 // MyArray class member- and friend-function definitions.
3 #include <algorithm>
4 #include <format>
5 #include <initializer_list>
6 #include <iostream>
7 #include <memory>
8 #include <span>
9 #include <sstream>
10 #include <stdexcept>
11 #include <utility>
12 #include "MyArray.h" // MyArray class definition
13
14 // MyArray constructor to create a MyArray of size elements containing 0
15 MyArray::MyArray(size_t size)
16   : m_size{size},
17   m_ptr{std::make_unique<int[]>(size)} {
18   std::cout << "MyArray(size_t) constructor\n";
19 }
20
```

Fig. 11.5 | MyArray class member- and friend-function definitions.



Checkpoint

I (True/False) The function `make_unique` _____ the dynamic memory it allocates.

Answer: value initializes.

11.6.5 Passing a Braced Initializer to a Constructor

In Fig. 6.2, we initialized a `std::array` object with a braced-initializer list, as in

```
std::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:

```
explicit MyArray(std::initializer_list<int> list);
```

The `std::initializer_list` class template is defined in `<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};
```

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 of Fig. 11.5) has one `initializer_list<int>` parameter named `list`. You can determine `list`'s number of elements by calling its **size member function** (line 23).

```
21 // MyArray constructor that accepts an initializer list
22 MyArray::MyArray(std::initializer_list<int> list)
23   : m_size{list.size()}, m_ptr{std::make_unique<int[]>(list.size())} {
24   std::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   std::copy(std::begin(list), std::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.



Checkpoint

- I (*True/False*) In a class that provides an `initializer_list` constructor, the class's other single-argument constructors must be called using parentheses rather than braces.
Answer: True.

11.6.6 Copy Constructor and Copy Assignment Operator

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**.¹⁵ 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 the standard library class `vector`, which uses RAII to manage resources.

CG

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 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,¹⁶ as we do in this case study. This is known as the **Rule of Five**.

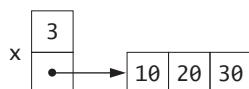
CG

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**.¹⁷ You also can explicitly remove compiler-generated special member functions to prevent the specified functionality by following their function prototypes with `= delete`. Class `unique_ptr` actually does this for the copy constructor and copy assignment operator.

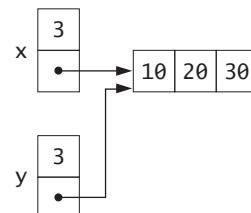
Shallow Copy

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`



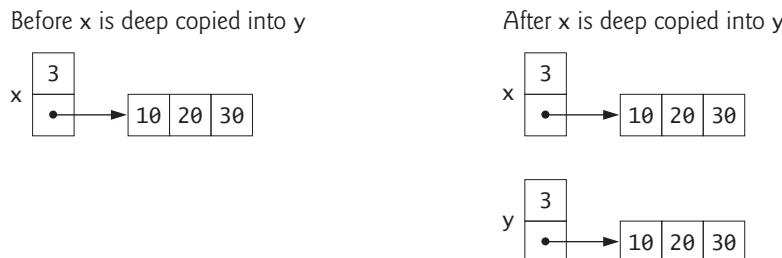
-
- 15. C++ Core Guidelines, “C.20: If You Can Avoid Defining Any Default Operations, Do.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero>.
 - 16. C++ Core Guidelines, “C.21: If You Define or =delete Any Copy, Move, or Destructor Function, Define or =delete Them All.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-five>.
 - 17. Scott Meyers, “A Concern About the Rule of Zero,” March 13, 2014. Accessed April 18, 2023. <http://scottmeyers.blogspot.com/2014/03/a-concern-about-rule-of-zero.html>.

Err ✗ Now, assume we'd like to copy `x` into a new object named `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**. Typically, this 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

In classes that manage their own resources, copying must be done carefully to avoid the pitfalls of shallow copies. 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.

Err ✗ 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.



Implementing the Copy Constructor

Line 21 of Fig. 11.4

```
MyArray(const MyArray& original); // copy constructor
```

SE 🚧 declares the class's **copy constructor** (defined lines 32–41). Its argument must be a **const reference** to prevent the constructor from modifying the argument object's data.

```

31 // copy constructor: must receive a reference to a MyArray
32 MyArray::MyArray(const MyArray& original)
33   : m_size{original.size()},
34   m_ptr{std::make_unique<int[]>(original.size())} {
35   std::cout << "MyArray copy constructor\n";
36
37   // copy original's elements into m_ptr's underlying int array
38   const std::span<const int> source{
39     original.m_ptr.get(), original.size()};
40   std::copy(std::begin(source), std::end(source), m_ptr.get());
41 }
42

```

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's size` member function.
- Line 34 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 35 outputs that the **copy constructor** was called.
- Recall that a **span** is a view into a contiguous collection of items, such as an array. Lines 38–39 create a `span` named `source` representing the argument `MyArray's dynamically allocated int array` from which we'll copy elements.
- Line 40 copies 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

```
MyArray& operator=(const MyArray& right); // copy assignment operator
```

declares the class's **overloaded copy assignment operator (=)**.¹⁸ This function's definition (lines 44–49) enables one `MyArray` to be assigned to another, copying the contents from the right operand into the left. When the compiler sees the statement

```
ints1 = ints2;
```

it invokes member function `operator=` with the call

```
ints1.operator=(ints2)
```

```
43 // copy assignment operator: implemented with copy-and-swap idiom
44 MyArray& MyArray::operator=(const MyArray& right) {
45     std::cout << "MyArray copy assignment operator\n";
46     MyArray temp{right}; // invoke copy constructor
47     swap(*this, temp); // exchange contents of this object and temp
48     return *this;
49 }
50
```

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**.^{19,20} The idiom operates as follows:

18. 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 (see Section 12.3).
19. Herb Sutter, “Exception-Safe Class Design, Part 1: Copy Assignment.” Accessed April 18, 2023. <http://www.gotw.ca/gotw/059.htm>.
20. Answer to “What is the copy-and-Swap Idiom?” Edited April 24, 2021, by Jack Lilhammers. Accessed April 18, 2023. <https://stackoverflow.com/a/3279550>.

- First, it copies the argument `right` into a local `MyArray` object (`temp`) using the **copy constructor** (line 46). If this fails to allocate memory for `temp`'s array, a **bad_alloc** exception will occur. In this case, the overloaded **copy assignment operator** will terminate without modifying the object on the assignment's left.
- Line 47 uses class `MyArray`'s **friend** function `swap` (defined in lines 169–172) 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 48), 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 46'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 49 of Fig. 11.3.



Checkpoint

1 *(Fill-in)* The C++ Core Guidelines recommend designing your classes so the compiler can autogenerate the five special member functions—known as the _____. Similarly, if you define any of these functions, you should define them all—known as the _____.

Answer: Rule of Zero, Rule of Five.

2 *(Fill-in)* The C++ Core Guidelines state that if a class requires one special member function, it should define them all. This is known as the _____.

Answer: Rule of Five.

3 *(Fill-in)* The compiler-generated copy constructor and copy assignment operator perform memberwise deep copies.

Answer: False. Actually, these perform memberwise shallow copies. Classes that manage their own resources should define their own copy constructor and overloaded copy assignment operator to perform deep copies.

11.6.7 Move Constructor and Move Assignment Operator



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:

```
MyArray(MyArray&& original) noexcept; // move constructor
MyArray& operator=(MyArray&& right) noexcept; // move assignment
```

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

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.²¹ Simply add `noexcept` after the function's signature in both the prototype and the definition. For a `const` member function, `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.

 Err

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 52–56) declares an *rvalue reference* (`&&`) parameter, indicating 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.

```

51 // move constructor: must receive an rvalue reference to a MyArray
52 MyArray::MyArray(MyArray&& original) noexcept
53     : m_size{std::exchange(original.m_size, 0)},
54     m_ptr{std::move(original.m_ptr)} { // move original.m_ptr into m_ptr
55     std::cout << "MyArray move constructor\n";
56 }
57

```

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, the object 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.²² To accomplish this for the `m_size` member, line 53

 SE

- 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` with the value that `exchange` returns.

When a `unique_ptr` is `move constructed`, as in line 54, 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`.²³ If your class manages raw pointers, you'd have to explicitly set the source pointer to `nullptr`—or use `exchange`, similar to line 53.

-
21. C++ Core Guidelines, “F.6: If Your Function May Not Throw, Declare It `noexcept`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-noexcept>.
 22. C++ Core Guidelines, “C.64: A Move Operation Should Move and Leave Its Source in a Valid State.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-semantic>.
 23. “`std::unique_ptr<T,Deleter>::unique_ptr`.” Accessed April 18, 2023. 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.²⁴

- 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 named `y`. Note that both members of `x` are 0 after the move—0 in a pointer member represents a null pointer.



Class MyArray’s Move Assignment Operator (=)

The move assignment operator (`=`) (lines 59–69) defines an *rvalue* reference (`&&`) parameter to indicate that its `MyArray` argument’s resources should be moved (not copied). Line 62 tests for `self-assignment` in which a `MyArray` object is being assigned to itself.²⁵ 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.

24. Topher Winward, “C++ Moves For People Who Don’t Know or Care What Rvalues Are,” January 17, 2019. Accessed April 18, 2023. <https://medium.com/@winwardo/c-moves-for-people-who-dont-know-or-care-what-rvalues-are-%EF%8F-56ee122dda7>.

25. “C.65: Make Move Assignment Safe for Self-Assignment.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self>.

```
58 // move assignment operator
59 MyArray& MyArray::operator=(MyArray&& right) noexcept {
60     std::cout << "MyArray move assignment operator\n";
61
62     if (this != &right) { // avoid self-assignment
63         // move right's data into this MyArray
64         m_size = std::exchange(right.m_size, 0); // indicate right is empty
65         m_ptr = std::move(right.m_ptr);
66     }
67
68     return *this; // enables x = y = z, for example
69 }
70
```

If it is not a **self-assignment**, lines 64–65

- 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`.

Move Operations Should Be noexcept

Move constructors and move assignment operators should never throw exceptions.



SE

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`.²⁶ This also is a requirement to be able to use your class's move capabilities with the standard library's containers like `vector`.



CG



Checkpoint

- 1 *(Fill-in)* A function declared _____ does not throw any exceptions and does not call any functions that throw exceptions. A function declared _____ indicates that the programmer has thought about whether the function might throw exceptions and has decided it might.

Answer: `noexcept`, `noexcept(false)`.

- 2 *(Fill-in)* A `unique_ptr`'s move constructor transfers ownership of the source `unique_ptr`'s dynamic memory to the target `unique_ptr` and sets the source `unique_ptr` to _____.

Answer: `nullptr`.

26. C++ Core Guidelines, “C.66: Make Move Operations `noexcept`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept>.

- 3 (Code) How would you declare an *rvalue* reference of type `std::string` for a function parameter named `value`?

Answer: `std::string&& value`.

11.6.8 Destructor

Line 27 of Fig. 11.4

```
~MyArray(); // destructor
```

declares the class's **destructor** (defined in lines 73–75), 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`.²⁷ 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.

```
71 // destructor: This could be compiler-generated. We included it here so
72 // we could output when each MyArray is destroyed.
73 MyArray::~MyArray() {
74     std::cout << "MyArray destructor\n";
75 }
76
```

11.6.9 `toString` and `size` Functions

Line 29 in Fig. 11.4

```
size_t size() const noexcept {return m_size;} // return size
```

defines an inline `size` member function, which returns a `MyArray`'s number of elements.

Line 30 in Fig. 11.4s

```
std::string toString() const; // create string representation
```

declares a `toString` member function (defined in lines 78–91), 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.

27. C++ Core Guidelines, “C.37: Make Destructors `noexcept`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-noexcept>.

```

77 // return a string representation of a MyArray
78 std::string MyArray::toString() const {
79     const std::span<const int> items{m_ptr.get(), m_size};
80     std::ostringstream output;
81     output << "{";
82
83     // insert each item in the dynamic array into the ostringstream
84     for (size_t count{0}; const auto& item : items) {
85         ++count;
86         output << item << (count < m_size ? ", " : "");
87     }
88
89     output << "}";
90     return output.str();
91 }
92

```

11.6.10 Overloading the Equality (==) and Inequality (!=) Operators

Line 33 of Fig. 11.4

```
bool operator==(const MyArray& right) const noexcept;
```

declares the **overloaded equality operator (==)**. Comparisons should not throw exceptions, so they should be declared noexcept.²⁸

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 95–101) operates as follows:

- Line 97 creates the span `lhs` representing the dynamically allocated `int` array in the left-hand-side operand (`ints1`).
- Line 98 creates the span `rhs` representing the dynamically allocated `int` array in the right-hand-side operand (`ints2`).
- Lines 99–100 use the standard library algorithm `equal` (from header `<algorithm>`) to compare corresponding elements from each span. 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`.

28. C++ Core Guidelines, “C.86: Make == Symmetric with Respect Of Operand Types and noexcept.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq>.

```

93 // determine if two MyArrays are equal and
94 // return true; otherwise, return false
95 bool MyArray::operator==(const MyArray& right) const noexcept {
96     // compare corresponding elements of both MyArrays
97     const std::span<const int> lhs{m_ptr.get(), size()};
98     const std::span<const int> rhs{right.m_ptr.get(), right.size()};
99     return std::equal(std::begin(lhs), std::end(lhs),
100                      std::begin(rhs), std::end(rhs));
101 }
102

```

Compiler Generates the != Operator

As of C++20, the compiler autogenerates 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 typically 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 Guideline

CG (●) C.164: Avoid Implicit Conversion Operators.²⁹ You may also come across the C++ Core Guideline "C.86: Make == Symmetric with Respect to Operand Types and noexcept,"³⁰

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.

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

```
bool operator==(const MyArray& left, const MyArray& right) noexcept;
```

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.

29. C++ Core Guidelines, "C.164: Avoid Implicit Conversion Operators," Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ro-conversion>.

30. C++ Core Guidelines, "C.86: Make == Symmetric with Respect to Operand Types and noexcept," Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq>.

C++ follows a complex set of overload-resolution rules to determine which function to call for each operator expression.³¹ 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

```
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.



Checkpoint

I *(Code)* Assuming class `MyArray` does not allow implicitly converting objects of other types to type `MyArray` or vice versa, rewrite `MyArray`'s `operator==` prototype as a member function of the class.

Answer: `bool operator==(const MyArray& right) noexcept;`

11.6.11 Overloading the Subscript ([]) Operator

Lines 36 and 39 of Fig. 11.4

```
int& operator[](size_t index);
const int& operator[](size_t index) const;
```

declare **overloaded subscript operators** (defined in lines 105–112 and 116–123). 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 116–123) 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

```
std::cout << z[3];
```

Remember that when an object is `const`, a program can invoke only the object's `const` member functions.

```
103 // overloaded subscript operator for non-const MyArrays;
104 // reference return creates a modifiable lvalue
105 int& MyArray::operator[](size_t index) {
106     // check for index out-of-range error
107     if (index >= m_size) {
108         throw std::out_of_range("Index out of range");
109     }
110
111     return m_ptr[index]; // reference return
112 }
113 }
```

(continued...)

31. “Overload Resolution.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/overload_resolution.

```

114 // overloaded subscript operator for const MyArrays
115 // const reference return creates a non-modifiable lvalue
116 const int& MyArray::operator[](size_t index) const {
117     // check for subscript out-of-range error
118     if (index >= m_size) {
119         throw std::out_of_range{"Index out of range"};
120     }
121
122     return m_ptr[index]; // returns copy of this element
123 }
124

```

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 111 and 122 belong to the 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.



Checkpoint

I (*Discussion*) Why does class `MyArray` provide the following two `operator[]` functions?

```

int& operator[](size_t index);
const int& operator[](size_t index) const;

```

Answer: These enable `[]` to be used with both `non-const` and `const` `MyArrays`. The `non-const` `operator[]` can be called only on `non-const` `MyArray` objects and enables getting or setting a specific element's value. The `const` version can be called only on `const` `MyArray` objects to get a specific element's value.

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

```
explicit operator bool() const noexcept {return size() != 0;}
```

defines an **inline** overloaded `operator bool` that converts a `MyArray` object to `true` if the `MyArray` is not empty or `false` if it is. 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 103 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.



Checkpoint

I (*Discussion*) An overloaded conversion operator does not specify a return type to the `operator` keyword's left. The return type is the _____ that defines the operator.

Answer: operator's type.

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 in lines 126–132, 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.

```
125 // preincrement every element, then return updated MyArray
126 MyArray& MyArray::operator++() {
127     // use a span and for_each to increment every element
128     const std::span<int> items{m_ptr.get(), m_size};
129     std::for_each(std::begin(items), std::end(items),
130                  [](auto& item){++item;});
131     return *this;
132 }
133
```

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. This enables a preincremented `MyArray` object to be used as an *lvalue*, which is how the built-in prefix increment operator works for fundamental types.



Checkpoint

1 (Code) What does the following code do?

```
MyArray& operator++();
```

Answer: This declares MyArray's unary overloaded preincrement operator (++).

2 (True/False) You can use the standard library's `for_each` algorithm to call a function that performs a task once for each element of a span.

Answer: True.

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 differentiates 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 135–139)

- 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 MyArray³²,
- 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).



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**.

```

I34 // postincrement every element, and return copy of original MyArray
I35 MyArray MyArray::operator++(int) {
I36     MyArray temp(*this);
I37     ++(*this); // call preincrement operator++ to do the incrementing
I38     return temp; // return the temporary copy made before incrementing
I39 }
I40

```

32. Herb Sutter, “GotW #2 Solution: Temporary Objects,” May 13, 2013. Accessed April 18, 2023.
<https://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/>.



Checkpoint

1 (*Discussion*) By convention, when the compiler sees a postincrement expression like `ints4++`, it generates the member-function call `ints4.operator++(0)`. Why does the compiler provide the 0 argument?

Answer: The 0 argument is strictly a dummy value that the compiler uses to distinguish between the prefix and postfix increment operator functions.

2 (*True/False*) The extra local object created by the prefix 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 postfix increment and decrement operators.

Answer: False. Actually, the **postfix** increment (or decrement) operator has this performance problem. For this reason, prefer the prefix increment and decrement operators.

11.6.15 Overloading the Addition Assignment Operator (+=)

Line 51 of Fig. 11.4

```
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 146) uses the `operator+=` function's `value` parameter in its body. The **lambda introducer** [`value`] specifies that the compiler should capture `value` for use in the lambda's body.

```
141 // add value to every element, then return updated MyArray
142 MyArray& MyArray::operator+=(int value) {
143     // use a span and for_each to increment every element
144     const std::span<int> items{m_ptr.get(), m_size};
145     std::for_each(std::begin(items), std::end(items),
146                  [value](auto& item) {item += value;});
147     return *this;
148 }
149
```

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

```
friend std::istream& operator>>(std::istream& in, MyArray& a);
```

and line 58 of Fig. 11.4

```
std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

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 152–160) takes as arguments an `istream` reference and a `MyArray` reference. It returns the `istream` reference argument to enable cascaded inputs, like

```
std::cin >> ints1 >> ints2;
```

When the compiler sees an expression like `cin >> ints1`, it invokes **non-member function `operator>>`** with the call

```
operator>>(std::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 153) representing `MyArray`'s dynamically allocated `int` array. Then lines 155–157 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.

```
150 // overloaded input operator for class MyArray;
151 // inputs values for entire MyArray
152 std::istream& operator>>(std::istream& in, MyArray& a) {
153     std::span<int> items{a.m_ptr.get(), a.m_size};
154
155     for (auto& item : items) {
156         in >> item;
157     }
158
159     return in; // enables cin >> x >> y;
160 }
161
```

Implementing the Stream Insertion Operator

The overloaded stream insertion function (`<<`) (lines 163–166) 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

```
std::cout << ints1 << ints2;
```

When the compiler sees an expression like `std::cout << ints1`, it invokes **non-member function `operator<<`** with the call

```
operator<<(std::cout, ints1)
```

```

162 // overloaded output operator for class MyArray
163 std::ostream& operator<<(std::ostream& out, const MyArray& a) {
164     out << a.toString();
165     return out; // enables std::cout << x << y;
166 }
167

```

Why `operator>>` and `operator<<` Must Be Non-member 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

```

std::cin >> ints4;
std::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 >> std::cin;
ints4 << std::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.

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.

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.³³



Another reason to 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

```

33. C++ Core Guidelines, “C.4: Make a Function a Member Only If It Needs Direct Access to the Representation of a Class.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-member>.

Each would sum an `int` value and a `HugeInt`. Like the built-in `+` operator for fundamental types, each would produce a temporary `HugeInt` containing the sum. To support these expressions, you define two versions of `operator+`, typically as non-member friend functions:

```
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.



Checkpoint

1 (*Fill-in*) The overloaded stream-extraction operator (`>>`) takes as one of its arguments an `istream` reference, which it returns to enable _____.

Answer: cascaded inputs.

2 (*Fill-in*) The overloaded stream insertion function (`<<`) receives as one of its arguments an `ostream` reference, which it returns to enable _____.

Answer: cascaded outputs.

3 (*Fill-in*) 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 _____—that interact with objects of the class through its `public` interface.

Answer: free functions.

11.6.17 friend Function swap

Line 13 of Fig. 11.4

```
friend void swap(MyArray& a, MyArray& b) noexcept;
```

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.³⁴ The function (lines 169–172) 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 each object's `m_ptr` members.



```
168 // swap function used to implement copy-and-swap copy assignment operator
169 void swap(MyArray& a, MyArray& b) noexcept {
170     std::swap(a.m_size, b.m_size); // swap using std::swap
171     a.m_ptr.swap(b.m_ptr); // swap using unique_ptr swap member function
172 }
```

11.7 C++20 Three-Way Comparison Operator (`<=>`)

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;

34. C++ Core Guidelines, “C.84: A swap Function May Not Fail.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-swap-fail>.

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 `<=` 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:

```
1 bool operator<=(const Time& right) const {
2     return !(right < *this);
3 }
```

This leads to lots of “boilerplate” code in which only the argument type differs in the definitions of the overloaded `!=`, `<=`, `>` and `<=` operators among classes.

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** (`$\langle=\rangle$`),^{35,36,37} which is also referred to as the spaceship operator³⁸ and requires the header `<compare>`. Figure 11.6 demonstrates `$\langle=\rangle$` for a class `Time` (lines 8–23), which contains

- a constructor (lines 10–11) to initialize its `private` data members (lines 20–22),
- a `toString` function (lines 13–15) to create a `Time`’s `string` representation and
- a **defaulted definition of the overloaded `$\langle=\rangle$ operator`** (line 18).

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’ element type.



```
1 // fig11_06.cpp
2 // C++20 three-way comparison (spaceship) operator.
3 Mapping Table Cell
4 #include <format>
5 #include <iostream>
6 #include <string>
7
8 class Time {
9 public:
10     Time(int hr, int min, int sec) noexcept
11         : m_hr{hr}, m_min{min}, m_sec{sec} {}
```

Fig. 11.6 | C++20 three-way comparison (spaceship) operator. (Part 1 of 2.)

-
35. “C++ Russia 2018: Herb Sutter, New in C++20: The Spaceship Operator,” YouTube Video, June 25, 2018. Accessed April 18, 2023. <https://www.youtube.com/watch?v=ULkwKsag0Yk>.
 36. Sy Brand, “Spaceship Operator,” August 23, 2018. Accessed April 18, 2023. <https://blog.tartan11ama.xyz/spaceship-operator/>.
 37. Cameron DaCamara, “Simplify Your Code with Rocket Science: C++20’s Spaceship Operator,” June 27, 2019. Accessed April 18, 2023. <https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/>.
 38. “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/W55xftItpl4/m/jcIttrMq8agJ?pli=1>.

```

12
13     std::string toString() const {
14         return std::format("hr={}, min={}, sec={}", m_hr, m_min, m_sec);
15     }
16
17     // <=> operator automatically supports equality/relational operators
18     auto operator<=>(const Time& t) const noexcept = default;
19 private:
20     int m_hr{0};
21     int m_min{0};
22     int m_sec{0};
23 };
24
25 int main() {
26     const Time t1(12, 15, 30);
27     const Time t2(12, 15, 30);
28     const Time t3(6, 30, 0);
29
30     std::cout << std::format("t1: {}\n"
31                           "t2: {}\n"
32                           "t3: {}\n", t1.toString(), t2.toString(), 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. (Part 2 of 2.)

In `main`, lines 26–28 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 34–46. 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 34–46 compiled and produced correct outputs, even though class `Time` did not define any equality or relational operators.

```
33 // using the equality and relational operators
34 std::cout << std::format("t1 == t2: {}\n", t1 == t2);
35 std::cout << std::format("t1 != t2: {}\n", t1 != t2);
36 std::cout << std::format("t1 < t2: {}\n", t1 < t2);
37 std::cout << std::format("t1 <= t2: {}\n", t1 <= t2);
38 std::cout << std::format("t1 > t2: {}\n", t1 > t2);
39 std::cout << std::format("t1 >= t2: {}\n\n", t1 >= t2);
40
41 std::cout << std::format("t1 == t3: {}\n", t1 == t3);
42 std::cout << std::format("t1 != t3: {}\n", t1 != t3);
43 std::cout << std::format("t1 < t3: {}\n", t1 < t3);
44 std::cout << std::format("t1 <= t3: {}\n", t1 <= t3);
45 std::cout << std::format("t1 > t3: {}\n", t1 > t3);
46 std::cout << std::format("t1 >= t3: {}\n\n", t1 >= t3);
47
```

```
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 49, 53 and 57.

```
48 // using <=> to perform comparisons
49 if ((t1 <=> t2) == 0) {
50     std::cout << "t1 is equal to t2\n";
51 }
52
53 if ((t1 <=> t3) > 0) {
54     std::cout << "t1 is greater than t3\n";
55 }
56
57 if ((t3 <=> t1) < 0) {
58     std::cout << "t3 is less than t1\n";
59 }
60 }
```

```
t1 is equal to t2
t1 is greater than t3
t3 is less than t1
```



Checkpoint

1 (*Fill-in*) For most types, the compiler can handle the comparison operators for you via the compiler-generated default implementation of C++20's _____.

Answer: three-way comparison operator ($\langle=\rangle$).

2 (*True/False*) In classes containing built-in array data members, the compiler applies operator $\langle=\rangle$ element-by-element for the objects' built-in array data members.

Answer: True.

3 (*Code*) Given the following strings

```
std::string s1{"hello"};
std::string s1{"goodbye"};
```

what is the value of the expression $s1 \langle=\rangle s2$?

Answer: This expression produces a positive value indicating that $s1$ is greater than $s2$.

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 *non-static 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

```
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.



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.



Checkpoint

- 1 *(Fill-in)* _____ constructors enable the compiler to convert among user-defined types or between user-defined types and fundamental types.

Answer: conversion.

- 2 *(True/False)* The compiler can use cast operators and conversion constructors implicitly to create objects.

Answer: True.

- 3 *(True/False)* C++ can satisfy an expression's needs by performing a series of implicit, user-defined conversions.

Answer: False. Actually, C++ can apply only one user-defined conversion per expression.

11.9 **explicit** Constructors and Conversion Operators

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.



SE

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.

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.



Err

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).

```

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    std::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

```

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 a `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 declared every single-argument constructor `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 class **MyArray** from Section 11.6, which included the keyword **explicit** in the declaration of its **single-argument constructor** that receives a **size_t**:

```
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

```
error: invalid initialization of reference of type ‘const MyArray&’  
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 to create a temporary **MyArray** of 3 elements and pass it to **outputArray**.

```
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    std::cout << "The MyArray received has " << arrayToOutput.size()  
19    << " elements. The contents are: " << arrayToOutput << "\n";  
20 }
```

Fig. 11.8 | Demonstrating an **explicit** constructor.

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.



SE

explicit Conversion Operators

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

```
explicit operator bool() const noexcept;
```



SE

declares the `bool` cast operator `explicit`, so you'd generally have to invoke it explicitly with `static_cast`, as in

```
static_cast<bool>(myArray0bject)
```

The `static_cast` operator explicitly converts its argument to the type in angle brackets. 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.



Checkpoint

1 (*True/False*) Any constructor that can be called with one argument and is not declared `explicit` can be used to perform an implicit conversion.

Answer: False. Copy and move constructors, which each receive one argument, do not perform implicit conversions.

2 (*Code*) What would you write before a single-argument constructor or a conversion operator to document that it can be used for implicit conversions?

Answer: `explicit(false)`.

3 (*Code*) What would you write before a single-argument constructor or a conversion operator to document that it cannot be used for implicit conversions?

Answer: `explicit` or `explicit(true)`.

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, we 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` is empty or contains 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. Finally, 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`.

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.

Exercises

11.1 (Memory Allocation and Deallocation Operators) Compare and contrast dynamic memory allocation and deallocation operators `new`, `new []`, `delete` and `delete []`.

11.2 (IntegerSet Class) Modify your `IntegerSet` class from Exercise 9.23 to reimplement member function `unionOfSets` as an overloaded `+` operator and `intersectionOfSets` as an overloaded `-` operator. Modify the application's `main` function to test your new operators.

11.3 (Complex Class) Consider the class `Complex` shown in Figs. 11.9–11.11, which enables operations on complex numbers of the form

`realPart + imaginaryPart * i`

where i is $\sqrt{-1}$.

- Modify the class to enable input and output of complex numbers via `>>` and `<<` operators.
- Overload the multiplication operator to multiply two complex numbers as in algebra.

After doing this exercise, you might want to read about the Standard Library's `complex` class (from header `<complex>`).

```

1 // Fig. 11.9: Complex.h
2 // Complex class definition.
3 #pragma once
4 #include <iostream>
5

```

Fig. 11.9 | Complex class definition. (Part I of 2.)

```

6  class Complex {
7  public:
8      explicit Complex(double realPart = 0.0, double imaginaryPart = 0.0);
9      Complex operator+(const Complex& right) const; // addition
10     Complex operator-(const Complex& right) const; // subtraction
11     std::string toString() const;
12 private:
13     double real; // real part
14     double imaginary; // imaginary part
15 };

```

Fig. 11.9 | Complex class definition. (Part 2 of 2.)

```

1 // Fig. 11.10: Complex.cpp
2 // Complex class member-function definitions.
3 #include <iostream>
4 #include <string>
5 #include "Complex.h" // Complex class definition
6
7 // Constructor
8 Complex::Complex(double realPart, double imaginaryPart)
9     : real{realPart}, imaginary{imaginaryPart} {}
10
11 // addition operator
12 Complex Complex::operator+(const Complex& operand2) const {
13     return Complex{real + operand2.real, imaginary + operand2.imaginary};
14 }
15
16 // subtraction operator
17 Complex Complex::operator-(const Complex& operand2) const {
18     return Complex{real - operand2.real, imaginary - operand2.imaginary};
19 }
20
21 // return string representation of a Complex object in the form: (a, b)
22 std::string Complex::toString() const {
23     return std::format("( {:.2f}, {:.2f})", real, imaginary);
24 }

```

Fig. 11.10 | Complex class member-function definitions.

```

1 // fig11_11.cpp
2 // Complex class test program.
3 #include <iostream>
4 #include <string>
5 #include "Complex.h"
6
7 int main() {
8     Complex x{};
9     Complex y{4.3, 8.2};
10    Complex z{3.3, 1.1};

```

Fig. 11.11 | Complex class test program. (Part I of 2.)

```

11
12
13     std::cout << std::format("x: {}\ny: {}\nz: {}\n\n",
14         x.toString(), y.toString(), z.toString());
15
16     x = y + z;
17     std::cout << std::format("{} = {} + {}\n\n",
18         x.toString(), y.toString(), z.toString());
19
20     x = y - z;
21     std::cout << std::format("{} = {} - {}\n",
22         x.toString(), y.toString(), z.toString());
23 }

```

```

x: (0.00, 0.00)
y: (4.30, 8.20)
z: (3.30, 1.10)

(7.60, 9.30) = (4.30, 8.20) + (3.30, 1.10)

(1.00, 7.10) = (4.30, 8.20) - (3.30, 1.10)

```

Fig. 11.11 | Complex class test program. (Part 2 of 2.)

11.4 (*HugeInteger Class*) A machine with 32-bit integers can represent integers in the range of approximately –2 billion to +2 billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. In Chapter 3’s Objects-Natural Case Study, we introduced super-sized integers via objects of the Boost Multiprecision library’s powerful `cpp_int` class. In this exercise, you’ll enhance the custom class `HugeInteger` shown in Figs. 11.12–11.14. This exercise is designed to give you practice overloading operators. For read-world applications, rather than reinventing the wheel, you should use the Boost Multiprecision library or another similar library for super-sized integers.

Study the `HugeInteger` class carefully, then respond to the following:

- Describe precisely how it operates.
- What restrictions does the class have?
- Overload the `*` multiplication operator.
- Overload the `/` division operator.
- Overload all the relational and equality operators.

[*Note:* We do not show an assignment operator or copy constructor for class `HugeInteger`, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

```

1 // Fig. 11.12: HugeInteger.h
2 // HugeInteger class definition.
3 #pragma once
4
5 #include <array>
6 #include <iostream>
7 #include <string>
8
9 class HugeInteger {
10     friend std::ostream& operator<<(std::ostream&, const HugeInteger&);
11 public:
12     static const int digits{40}; // maximum digits in a HugeInteger
13
14     HugeInteger(long long value = 0); // conversion/default constructor
15     HugeInteger(const std::string& value); // conversion constructor
16
17     // addition operator; HugeInteger + HugeInteger
18     HugeInteger operator+(const HugeInteger& other) const;
19
20     // addition operator; HugeInteger + int
21     HugeInteger operator+(long long other) const;
22
23     // addition operator;
24     // HugeInteger + string that represents large integer value
25     HugeInteger operator+(const std::string& other) const;
26 private:
27     std::array<short, digits> integer{}; // default init to 0s
28 };

```

Fig. 11.12 | HugeInteger class definition.

```

1 // Fig. 11.13: HugeInteger.cpp
2 // HugeInteger member-function and friend-function definitions.
3 #include <cctype> // isdigit function prototype
4 #include "HugeInteger.h" // HugeInteger class definition
5
6 // default constructor; conversion constructor that converts
7 // a long long into a HugeInteger object
8 HugeInteger::HugeInteger(long long value) {
9     // place digits of argument into array
10    for (int j{digits - 1}; value != 0 && j >= 0; j--) {
11        integer[j] = value % 10;
12        value /= 10;
13    }
14 }
15

```

Fig. 11.13 | HugeInteger member-function and friend-function definitions. (Part 1 of 3.)

```

16 // conversion constructor that converts a character string
17 // representing a large integer into a HugeInteger object
18 HugeInteger::HugeInteger(const std::string& value) {
19     // place digits of argument into array
20     size_t length{value.size()};
21
22     for (size_t j{digits - length}, k{0}; j < digits; ++j, ++k) {
23         if (std::isdigit(value[k])) { // ensure that character is a digit
24             integer[j] = value[k] - '0';
25         }
26     }
27 }
28
29 // addition operator; HugeInteger + HugeInteger
30 HugeInteger HugeInteger::operator+(const HugeInteger& other) const {
31     HugeInteger temp; // temporary result
32     int carry{0};
33
34     for (int i{digits - 1}; i >= 0; i--) {
35         temp.integer[i] = integer[i] + other.integer[i] + carry;
36
37         // determine whether to carry a 1
38         if (temp.integer[i] > 9) {
39             temp.integer[i] %= 10; // reduce to 0-9
40             carry = 1;
41         }
42         else { // no carry
43             carry = 0;
44         }
45     }
46
47     return temp; // return copy of temporary object
48 }
49
50 // addition operator; HugeInteger + int
51 HugeInteger HugeInteger::operator+(long long other) const {
52     // convert other to a HugeInteger, then invoke
53     // operator+ for two HugeInteger objects
54     return *this + HugeInteger(other);
55 }
56
57 // addition operator;
58 // HugeInteger + string that represents large integer value
59 HugeInteger HugeInteger::operator+(const std::string& other) const {
60     // convert other to a HugeInteger, then invoke
61     // operator+ for two HugeInteger objects
62     return *this + HugeInteger(other);
63 }
64
65 // overloaded output operator
66 std::ostream& operator<<(std::ostream& output, const HugeInteger& num) {
67     size_t i;
68

```

Fig. 11.13 | HugeInteger member-function and friend-function definitions. (Part 2 of 3.)

```
69     // skip leading zeros
70     for (i = 0; i < HugeInteger::digits && 0 == num.integer[i]; ++i) { }
71
72     if (i == HugeInteger::digits) {
73         output << 0;
74     }
75     else {
76         for (; i < HugeInteger::digits; ++i) {
77             output << num.integer[i];
78         }
79     }
80 }
81
82 return output;
}
```

Fig. 11.13 | HugeInteger member-function and friend-function definitions. (Part 3 of 3.)

Fig. 11.14 | HugeInteger test program. (Part 1 of 2.)

$$7654321 + 9 = 7654330$$

$$7891234 + 10000 = 7901234$$

Fig. 11.14 | HugeInteger test program. (Part 2 of 2.)

11.5 (Fraction Class) Create a Fraction class like the one in Exercise 9.16. Provide the following capabilities:

- a) Create a constructor that prevents a 0 denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.
 - b) Overload the addition, subtraction, multiplication and division operators for two Fractions.
 - c) Overload the relational and equality operators for two Fractions.

11.6 (Polynomial Class) Develop class `Polynomial`. A `Polynomial`'s internal representation is an array of terms. Each term contains a coefficient and an exponent. For example, the term

$$2x^4$$

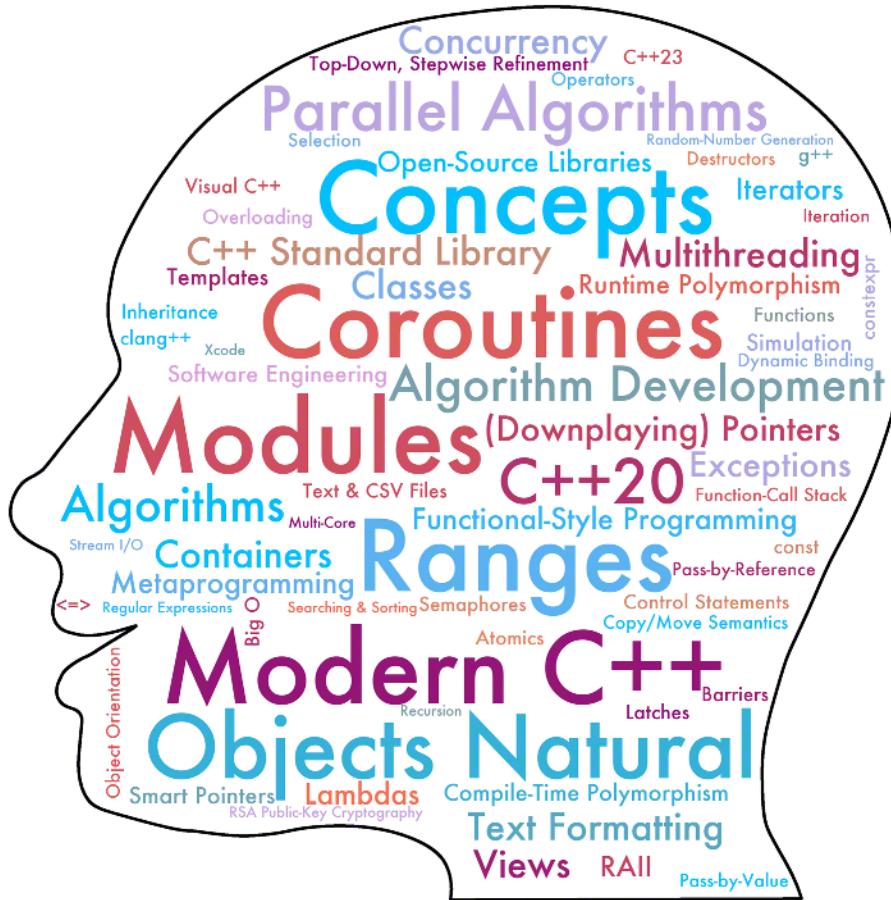
has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as *set* and *get* functions. The class should also provide the following overloaded operator capabilities:

- a) Overload the addition operator (+) to add two Polynomials.
 - b) Overload the subtraction operator (-) to subtract two Polynomials.
 - c) Overload the assignment operator to assign one Polynomial to another.
 - d) Overload the multiplication operator (*) to multiply two Polynomials.
 - e) Overload the addition assignment operator (+=), subtraction assignment operator (-=), and multiplication assignment operator (*=).

This page intentionally left blank

Exceptions and a Look Forward to Contracts

12



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 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	12.7 Constructors, Destructors and Exception Handling
12.2 Exception-Handling Flow of Control	12.7.1 Throwing Exceptions from Constructors
12.2.1 Defining a Custom Exception Class	12.7.2 Catching Exceptions in Constructors via Function <code>try</code> Blocks
12.2.2 Demonstrating Exception Handling	12.7.3 Exceptions and Destructors: Revisiting <code>noexcept(false)</code>
12.2.3 Enclosing Code in a <code>try</code> Block	
12.2.4 Defining a <code>catch</code> Handler for <code>DivideByZeroExceptions</code>	12.8 Processing <code>new</code> Failures
12.2.5 Termination Model of Exception Handling	12.8.1 <code>new</code> Throwing <code>bad_alloc</code> on Failure
12.2.6 Flow of Control When the User Enters a Nonzero Denominator	12.8.2 <code>new</code> Returning <code>nullptr</code> on Failure
12.2.7 Flow of Control When the User Enters a Zero Denominator	12.8.3 Handling <code>new</code> Failures Using Function <code>set_new_handler</code>
12.3 Exception Safety Guarantees and <code>noexcept</code>	12.9 Standard Library Exception Hierarchy
12.4 Rethrowing an Exception	12.10 C++'s Alternative to the <code>finally</code> Block: Resource Acquisition Is Initialization (RAII)
12.5 Stack Unwinding and Uncaught Exceptions	12.11 Some Libraries Support Both Exceptions and Error Codes
12.6 When to Use Exception Handling	12.12 Logging
12.6.1 <code>assert</code> Macro	12.13 Looking Ahead to Contracts
12.6.2 Failing Fast	12.14 Wrap-Up Exercises

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:

- 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

-
1. Bjarne Stroustrup, "C++ Applications," October 27, 2020. Accessed April 18, 2023. <https://www.stroustrup.com/applications.html>.
 2. Vincent Lextrait, "The Programming Languages Beacon v16," March 2016. Accessed April 18, 2023. <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

- 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.
- Facebook has 62 million lines of code.
- An average modern high-end car has 100 million lines of code, and they’re expected to have 300 million lines of code by 2030⁵—across hundreds of processors and controllers.⁶

Self-driving cars are expected to require one billion lines of code.⁷ For any sized codebase, it’s essential to eliminate bugs during development and handle problems that may occur once the software is deployed in real products. That’s the focus of this chapter.

Exceptions and Exception Handling

As you know, an **exception** indicates 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.

-
3. “Codebases: Millions of Lines of Code Infographic,” September 24, 2015. Accessed April 18, 2023. <https://www.informationisbeautiful.net/visualizations/million-lines-of-code/>. For a spreadsheet of the infographic’s data sources, see <http://bit.ly/CodeBasesInfographicData>.
 4. “The Large Hadron Collider.” Accessed April 18, 2023. <https://home.cern/science/accelerators/large-hadron-collider>.
 5. Anthony Martin, “Vehicle Cybersecurity: Control the Code, Control the Road,” March 18, 2020. Accessed April 18, 2023. <https://www.vehicledynamicsinternational.com/features/vehicle-cybersecurity-control-the-code-control-the-road.html>.
 6. Ferenc Valenta’s answer to “How many lines of code are in a car?” January 10, 2019. Accessed April 18, 2023. <https://www.quora.com/How-many-lines-of-code-are-in-a-car/answer/Ferenc-Valenta>.
 7. “Jaguar Land Rover Finds the Teenagers Writing the Code for a Self-Driving Future,” April 15, 2019. Accessed April 18, 2023. <https://media.jaguarlandrover.com/news/2019/04/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future>.

Exception Handling, Stack Unwinding and Rethrowing Exceptions

You've seen exceptions get thrown (Section 6.15) and `try...catch` used to handle them (Section 9.7.11). 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, `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 derived-class exception types.

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.

Looking Ahead to Contracts

The chapter concludes with an introduction to contracts. This feature was originally targeted for C++20 but delayed to a future version. We'll introduce preconditions, postconditions and assertions, which you'll see are implemented as contracts tested at execution time. If such conditions fail, a contract violation occurs. By default, the code terminates

8. Herb Sutter, "De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable," September 23, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=ARYP83yNAWk>.

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 GCC's experimental contracts implementation on <https://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

SE

Section 12.5, you'll see that ignoring an exception can lead to program termination.^{12,13}

SE

Indicating Errors via Return Values

Without exception handling, it's common for a function to calculate and return a value Err 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.



Checkpoint

1 *(True/False)* Exceptions may surface through code in a `try` block, calls to other functions and operator errors, like `new` failing to acquire additional memory at execution time.

Answer: True.

2 *(True/False)* C++ enables you to ignore exceptions to enhance program performance.

Answer: False. Actually, once an exception occurs, ignoring it leads to program termination.

3 *(Discussion)* Without exception handling, it's common for a function to calculate and return a value on success or return an error indicator on failure. Discuss the problems with this architecture.

Answer: One problem is using the return value in a subsequent calculation without first checking whether the value is the error indicator. Also, returning error codes is not always possible, such as in constructors and overloaded operators.

9. "Technical Report on C++ Performance," February 15, 2006. Accessed April 18, 2023. <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 April 18, 2023. <https://isocpp.org/wiki/faq/exceptions#exceptions-separate-good-and-bad-path>.

11. Barbara Thompson, "C++ Exception Handling: Try, Catch, Throw Example," updated January 1, 2022. Accessed April 18, 2023. <https://www.guru99.com/cpp-exceptions-handling.html>.

12. "Technical Report on C++ Performance," February 15, 2006. Accessed April 18, 2023. <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 April 18, 2023. <https://dzone.com/articles/some-useful-facts-to-know-when-using-c-exceptions>.

12.2 Exception-Handling Flow of Control

Let's demonstrate the flow of control:

- when a program executes successfully and
- when an exception occurs.

For demo purposes, Figs. 12.1–12.2 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** representing the type of problem that might occur in the example.
- `fig12_02.cpp` (Fig. 12.2) 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 a Custom Exception Class

Figure 12.1 defines `DivideByZeroException`—a custom exception class 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>`). 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.

```

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};

```

Fig. 12.1 | Class `DivideByZeroException` definition.



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.



Checkpoint

1 (*Fill-in*) A typical derived class of `runtime_error` defines only a(n) _____.

Answer: constructor that passes an error-message string to the base-class constructor.

2 (*Fill-in*) Generally, you should derive a custom exception class from a(n) _____ and choose a name that makes it clear what problem occurred.

Answer: C++ standard library exception class.

12.2.2 Demonstrating Exception Handling

Figure 12.2 uses exception handling to wrap code that might throw a `DivideByZeroException` and to handle that exception should one occur. The `quotient` function receives two doubles, divides the first by the second and returns the double result. The function treats all attempts to divide by zero as errors. If the second argument is zero, it throws a `DivideByZeroException` (line 13) to indicate this problem to the caller. The `throw` operator can be an object of any copy-constructible type, not just `exception` and its derived types. Copy-constructible types are required because the exception mechanism copies the exception into a temporary exception object.¹⁵

```

1 // fig12_02.cpp
2 // Example that throws an exception on
3 // an attempt to divide by zero.
4 #include <format>
5 #include <iostream>
6 #include "DivideByZeroException.h" // DivideByZeroException class
7
8 // performs division only if the denominator is not zero;
9 // otherwise, throws DivideByZeroException object
10 double quotient(double numerator, double denominator) {
11     // throw DivideByZeroException if trying to divide by zero
12     if (denominator == 0.0) {
13         throw DivideByZeroException{};
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 }
```

Fig. 12.2 | Example that throws an exception on an attempt to divide by zero. (Part I of 2.)

-
- 14. C++ Core Guidelines, “E.14: Use Purpose-Designed User-Defined Types as Exceptions (not Built-in Types).” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-types>.
 - 15. “throw Expression.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/throw>.

```

24     std::cout << "Enter two integers (end-of-file to end): ";
25
26     // enable user to enter two integers to divide
27     while (std::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             std::cout << std::format("The quotient is: {:.2f}\n", result);
33         }
34         catch (const DivideByZeroException& divideByZeroException) {
35             std::cout << std::format("Exception occurred: {}\n",
36                                     divideByZeroException.what());
37         }
38
39         std::cout << "\nEnter two integers (end-of-file to end): ";
40     }
41
42     std::cout << '\n';
43 }
```

Enter two integers (end-of-file to end): 100 7
 The quotient is: 14.29

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 the user does not specify 0 as the denominator for the division, `quotient` returns the division result. If the user enters 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 improper input, **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.

Checkpoint

- I (Fill-in) The operand of a `throw` can be of any _____ type, not just objects of types that directly or indirectly derive from `exception`.

Answer: copy-constructible

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



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 to the exception type to process—in this case, a `DivideByZeroException`. This has two key benefits:



- It prevents copying the exception as part of the `catch` operation.
- It enables catching derived-class exceptions without slicing (see below).



catching By Reference Avoids Slicing

When an exception occurs in a `try` block, C++ executes the first `catch` handler with a parameter of the same type as, or a base-class type of, the thrown exception's type. 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 logic error, known as **slicing**, occurs when a derived-class object is copied into or assigned to a base-class object. Always catch exceptions by reference to prevent slicing.



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. Line 36 calls its `what` member function to get the exception's error message.

Tasks Performed in catch Handlers

Typical tasks performed by `catch` handlers include

- logging errors to a file that developers can study for debugging purposes,
- terminating the program gracefully,
- trying an alternative strategy to accomplish the failed task,
- rethrowing the exception to the current function's caller or
- throwing a different exception type.

This example's `catch` handler simply reports that the user attempted to divide by zero for demonstration purposes.

16. C++ Core Guidelines, “E.2: Throw an Exception to Signal That a Function Can't Perform Its Assigned Task.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-throw>.

Common Errors When Defining catch Handlers

Programmers new to exception handling should be aware of several common coding errors:



- It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.



- Each `catch` handler can have only one parameter—specifying a comma-separated list of exception parameters is a syntax error.



- It's a compilation error to catch the same type in multiple `catch` handlers following a single `try` block.



- It's a logic error to catch a base-class exception before a derived-class exception—compilers typically warn you when this occurs.

Checkpoint

1 *(Fill-in)* Declare each exception parameter as a(n) _____ to prevent copying the exception object and to enable catching derived-class exceptions.

Answer: const reference to the exception type to handle.

2 *(Fill-in)* When an exception occurs in a `try` block, C++ executes the first `catch` handler with a parameter of the same type as, or a(n) _____ of, the thrown exception's type.

Answer: base-class type

3 *(Fill-in)* The logic error known as _____ occurs when a derived-class object is copied into or assigned to a base-class object.

Answer: slicing.

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, enabling programs to avoid resource leaks.¹⁷ 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.

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.

17. "Technical Report on C++ Performance," February 15, 2006. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf> (page 34).

Checkpoint

- 1** (*Fill-in*) When an exception occurs in a `try` block, the `try` block terminates. Any local variables defined in that block go out of scope, and their destructors execute, enabling programs to avoid _____.

Answer: resource leaks.

- 2** (*Fill-in*) If an exception occurs in a function but is not caught there, the function terminates immediately. The program attempts to locate an enclosing `try` block in the calling function. This process is called _____.

Answer: stack unwinding

- 3** (*True/False*) If no exceptions occur in a `try` block, program control continues with the first statement after that `try` block.

Answer: False. Actually, program control continues with the first statement after the last catch that follows the `try` block.

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 a `throw` statement 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) to the constructor as you create the object, as in

```
throw out_of_range{"Index out of range"};
```

When the exception is thrown, `quotient` exits immediately without performing 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 that encounters an error and throws an exception.

We enclosed the `quotient` call (line 31) in a `try` block, so 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.



Checkpoint

- 1** (*True/False*) Generally, you throw an exception immediately after the error occurs.

Answer: False. Actually, if you explicitly throw an exception, you generally do so before the error has an opportunity to occur.

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** you'll make:^{18,19}

- **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 before 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, move assignment operator and friend function swap 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 a noexcept function, the program terminates immediately.



Checkpoint

- 1** (*True/False*) With no exception safety guarantee, if an exception occurs, the program may be in an invalid state and resources like dynamically allocated memory could leak.

Answer: True.

- 2** (*True/False*) With a basic exception guarantee, if an exception occurs, objects' states remain valid but possibly modified, and no resources leak.

Answer: True.

- 3** (*True/False*) With a strong exception guarantee, if an exception occurs, objects' states remain unmodified or return to their original states before the operation that caused the exception.

Answer: True.

18. Klaus Iglberger, "Back to Basics: Exceptions," October 5, 2020. Accessed April 18, 2023 <https://www.youtube.com/watch?v=0ojB8c0xUd8>.

19. "Exceptions." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/exceptions>.

12.4 Rethrowing an Exception

Sometimes, you might

- catch an exception,
- partially process it, then
- notify the caller by **rethrowing the exception**.

Doing so indicates that the exception is not yet handled. The statement

`throw;`

executed in a catch handler returns the current exception to the next enclosing try block. Then a catch handler listed after that try block attempts to handle the exception. Executing the preceding statement outside a catch handler terminates the program immediately.



Err

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.
- 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 `nested_exception` class.²⁰
- 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 24–28), line 26 calls `throwException` (lines 7–20), which contains a try block (lines 9–12) from which the `throw` statement in line 11 throws an `exception` object. The function's catch handler (lines 13–17) catches this exception, prints an error message (lines 14–15) and rethrows the exception (line 16). This immediately returns control to line 26 in `main`'s try block. That try block *terminates* (so line 27 does *not* execute), and the catch handler in `main` (lines 29–31) catches the exception and prints an error message (line 30). We do not use the exception parameters in this example's catch handlers, so we omit the exception parameter names and specify only the type of exception to catch (lines 13 and 29).

20. “`std::nested_exception`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/error/nested_exception.

```

1 // fig12_03.cpp
2 // Rethrowing an exception.
3 #include <iostream>
4 #include <exception>
5
6 // throw, catch and rethrow exception
7 void throwException() {
8     // throw exception and catch it immediately
9     try {
10         std::cout << " Function throwException throws an exception\n";
11         throw std::exception{}; // generate exception
12     }
13     catch (const std::exception&) { // handle exception
14         std::cout << " Exception handled in function throwException"
15             << "\n Function throwException rethrows exception";
16         throw; // rethrow exception for further processing
17     }
18
19     std::cout << "This should not print\n";
20 }
21
22 int main() {
23     // call throwException
24     try {
25         std::cout << "main invokes function throwException\n";
26         throwException();
27         std::cout << "This should not print\n";
28     }
29     catch (const std::exception&) { // handle exception
30         std::cout << "\n\nException handled in main\n";
31     }
32
33     std::cout << "Program control continues after catch in main\n";
34 }
```

```

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.



Checkpoint

1 (Fill-in) Rethrowing an exception indicates that _____.

Answer: the exception is not yet handled.

2 (Fill-in) In a catch handler, 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 _____ class.

Answer: nested_exception.

- 3 (Code) Write a statement that you'd place in a `catch` handler to return the current exception to the next enclosing `try` block.

Answer: `throw;`

12.5 Stack Unwinding and Uncaught Exceptions

When you do not catch an exception in a particular scope, the function-call stack “unwinds” in an attempt to catch the exception in an **enclosing scope**—that is, a `catch` block of the next outer `try` block. 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 you should not wrap a `try...catch` around every function call that might throw an exception.²¹ Sometimes it's more appropriate to let an earlier function in the call chain deal with the problem. Figure 12.4 demonstrates stack unwinding.



```
1 // fig12_04.cpp
2 // Demonstrating stack unwinding.
3 #include <iostream>
4 #include <stdexcept>
5
6 // function3 throws runtime error
7 void function3() {
8     std::cout << "In function 3\n";
9
10    // no try block, stack unwinding occurs, return control to function2
11    throw std::runtime_error{"runtime_error in function3"};
12 }
13
14 // function2 invokes function3
15 void function2() {
16     std::cout << "function3 is called inside function2\n";
17     function3(); // stack unwinding occurs, return control to function1
18 }
19
20 // function1 invokes function2
21 void function1() {
22     std::cout << "function2 is called inside function1\n";
23     function2(); // stack unwinding occurs, return control to main
24 }
25
```

Fig. 12.4 | Demonstrating stack unwinding. (Part 1 of 2.)

21. C++ Core Guidelines, “E.17: Don't Try to Catch Every Exception in Every Function.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-not-always>.

```

26 // demonstrate stack unwinding
27 int main() {
28     // invoke function1
29     try {
30         std::cout << "function1 is called inside main\n";
31         function1(); // call function1 which throws runtime_error
32     }
33     catch (const std::runtime_error& error) { // handle runtime error
34         std::cout << "Exception occurred: " << error.what()
35             << "\nException handled in main\n";
36     }
37 }
```

```

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 31 in the `try` block calls `function1` (lines 21–24), then `function1` calls `function2` (lines 15–18), which in turn calls `function3` (lines 7–12). Line 11 of `function3` throws a `runtime_error` object—this is the throw point. At this point, control proceeds as follows:

- No `try` block encloses line 11, so stack unwinding begins. `function3` terminates, returning control to line 17 in `function2`.
- No `try` block encloses line 17, so stack unwinding continues. `function2` terminates, returning control to line 23 in `function1`.
- Again, no `try` block encloses line 23, so stack unwinding occurs again. `function1` terminates and returns control to line 31 in `main`.
- The `try` block in lines 29–32 encloses line 31, so the `try` block's first matching `catch` handler (lines 33–36) catches and processes the exception by displaying the exception message.

Uncought Exceptions



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 30–31 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 and the exception is not caught in `main`, the program terminates. The following output was produced when we compiled with GNU C++ then ran the program. Note the mention that `terminate` was called:

```
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)
```



Checkpoint

- 1 (*Fill-in*) When you do not catch an exception in a particular scope, the function-call stack _____ in an attempt to catch the exception in an enclosing scope—that is, a catch block of the next outer `try` block.

Answer: unwinds.

- 2 (*Fill-in*) If an exception is not caught during stack unwinding, C++ calls the standard library function _____, which calls `abort` to terminate the program.

Answer: `terminate`.

- 3 (*True/False*) You should wrap a `try...catch` around every function call that might throw an exception.

Answer: False. Actually, sometimes it's more appropriate to let the stack unwind so an earlier function in the call chain can deal with the problem.

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 correct,²² such as

- 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 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.

22. "Modern C++ Best Practices for Exceptions and Error Handling." Accessed April 18, 2023. <https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp>.

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 inception²³—doing so after a system has been implemented can be difficult.

 SE

 CG

When Not to Use Exception Handling

The isocpp.org FAQ section on exceptions lists several scenarios in which you should *avoid* using exceptions.²⁴ These include



- **cases in which failures are expected**, such as converting incorrectly formatted strings to numeric values where the strings might not have the correct format;
- **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;²⁵ and
- **frequent errors that should not happen in code**, such as **accessing out-of-range array elements**, **dereferencing a null pointer** and **division by zero**.²⁶ Interestingly, 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 program,²⁷ which may be **unacceptable in memory-constrained devices, such as embedded systems**. According to the isocpp.org 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



Functions with common errors 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 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 run-time 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

23. C++ Core Guidelines, “E.1: Develop an Error-Handling Strategy Early in a Design.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-design>.

24. “What Shouldn’t I Use Exceptions For?” Accessed April 18, 2023. <https://isocpp.org/wiki/faq/exceptions#why-not-exceptions>.

25. “Joint Strike Fighter Air Vehicle C++ Coding Standards,” December 2005. Accessed April 18, 2023. <https://www.stroustrup.com/JSF-AV-rules.pdf>.

26. We used division by zero as a mechanical demonstration of the exception handling flow of control.

27. Vishal Chovatiya, “C++ Exception Handling Best Practices: 7 Things To Know,” November 3, 2019. Accessed April 18, 2023. <http://www.vishalchovatiya.com/7-best-practices-for-exception-handling-in-cpp-with-example/>.

28. “Why Use Exceptions?” Accessed April 18, 2023. <https://isocpp.org/wiki/faq/exceptions#why-exceptions>.

for such errors.²⁹ 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.

The C++ Core Guidelines indicate that many `try...catch` statements in your code can be a sign of too much low-level resource management.³⁰ To avoid this, they recommend designing your classes to use RAI^I (introduced in Section 11.5) so an object's constructor acquires resources and its destructor releases them. So, if an object goes out of scope for any reason, including an exception, the object's resources will be released.



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** from the `<cassert>` header,³¹ as in

```
assert(condition);
```

Assertions are primarily a development-time aid³² for alerting programmers to coding errors that need to be fixed. 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:

```
#define NDEBUG
```

Compilers typically provide a setting that enables you to disable assertions without having to modify your code. For example, `g++` allows the command-line option `-DNDEBUG`.

12.6.2 Failing Fast

The C++ Core Guidelines suggest that "If you can't throw exceptions, consider failing fast."³³ **Fail-fast** 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.³⁴ This seems counterintuitive. The idea is that failing fast actually



-
29. Herb Sutter, "Zero-Overhead Deterministic Exceptions: Throwing Values," August 4, 2019. Accessed April 18, 2023. <https://wg21.link/p0709R4>.
 30. C++ Core Guidelines, "E.18: Minimize the Use of Explicit `try/catch`." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-catch>.
 31. "assert." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/error/assert>.
 32. "Modern C++ Best Practices For Exceptions and Error Handling," August 24, 2020. Accessed April 18, 2023. <https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp>.
 33. C++ Core Guidelines, "E.26: If You Can't Throw Exceptions, Consider Failing Fast." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw-crash>.
 34. "Fail-Fast." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. <https://en.wikipedia.org/wiki/Fail-fast>.

helps you build more robust software by finding and fixing errors sooner during development. Fewer errors are likely to make their way into the final product.³⁵



Checkpoint

1 (*Fill-in*) _____ 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.

Answer: Fail-fast.

2 (*Fill-in*) _____ help ensure a program's validity by catching potential bugs and identifying possible logic errors during development.

Answer: Assertions.

3 (*True/False*) Functions with common errors should always throw exceptions.

Answer: False. Actually, functions with common errors typically should return `nullptr`, 0 or other appropriate values, such as `bools`, rather than throw exceptions.

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 initializers and
- why destructors should not throw exceptions.

12.7.1 Throwing Exceptions from Constructors

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? A **constructor cannot return a value to indicate an error**, so how can we indicate that the object was not 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. However, do not throw exceptions from the constructor of a global object. Such exceptions cannot be caught because they're constructed *before* `main` executes.



A constructor should throw an exception if a problem occurs while initializing an object. If the constructor manages dynamically allocated memory without smart pointers, it should release that memory before throwing an exception to prevent memory leaks. The destructor will not be called to release resources, though destructors for data members that have already been constructed will be called. **Always use smart pointers to manage dynamically allocated memory.**



35. 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>.



Checkpoint

1 *(Discussion)* How can a constructor indicate that an error occurred?

Answer: Since a constructor cannot return a value to indicate an error, a constructor can throw an exception that contains the error information.

2 *(Discussion)* What should a constructor do when an error occurs if the constructor allocates memory without using smart pointers?

Answer: Since the destructor will not be called to release resources, the constructor should release the memory before throwing an exception .

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.

```
1 // fig12_05.cpp
2 // Demonstrating a function try block.
3 #include <format>
4 #include <iostream>
5 #include <limits>
6 #include <stdexcept>
7
8 // class Integer purposely throws an exception from its constructor
9 class Integer {
10 public:
11     explicit Integer(int i) : value{i} {
12         std::cout << std::format("Integer constructor: {}\n", value)
13             << "Purposely throwing exception from Integer constructor\n";
14         throw std::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         std::cout << "ResourceManager constructor called\n";
24     }
25     catch (const std::runtime_error& ex) {
26         std::cout << std::format(
27             "Exception while constructing ResourceManager: {}", ex.what()
28             << "\nAutomatically rethrowing the exception\n";
29     }
30 private:
31     Integer myInteger;
32 };
33
```

Fig. 12.5 | Demonstrating a function try block. (Part I of 2.)

```

34 int main() {
35     try {
36         const ResourceManager resource{7};
37     }
38     catch (const std::runtime_error& ex) {
39         std::cout << std::format("Rethrown exception caught in main: {}\n",
40                               ex.what());
41     }
42 }
```

```

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
```

Fig. 12.5 | Demonstrating a function `try` block. (Part 2 of 2.)

To help demonstrate a function `try` block, class `Integer` (lines 9–18) simulates failing to “acquire a resource” by purposely throwing an exception (line 14) from its constructor. Class `ResourceManager` (lines 20–32) contains an object of class `Integer` (line 31), 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 exception that occurs 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 25–29. The flow of control in this example is as follows:

- Line 36 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 the `Integer` class’s constructor and throws the exception back to the 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 25–29 catches the exception and displays the next two lines of output.
- 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.³⁶ Our catch handler does not explicitly contain a `throw` statement, so it implicitly rethrows the exception. This terminates the `ResourceManager` constructor and throws the exception back to line 36 in `main`.

- Line 36 is in a `try` block, so that block terminates, and the catch handler in lines 38–41 handles the exception, displaying the last line of the output.

Function `try` blocks also may be used with other functions using the following syntax:  **SE**

```
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:

```
void myFunction() {
    try {
        // do something
    }
    catch (const ExceptionType& ex) {
        // exception processing
    }
}
```



Checkpoint

I (*True/False*) 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.

Answer: True.

12.7.3 Exceptions and Destructors: Revisiting `noexcept(false)`

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)`.

If your destructor calls functions that might throw exceptions, you should catch and handle them, even if that simply means logging the exception and terminating the program in a controlled manner.³⁷ 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.  **SE**

36. “Function-try-block.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/function-try-block>.

37. C++ Core Guidelines, “C.36: A Destructor Must Not Fail.” Accessed April 18, 2023. <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 or base-class subobjects constructed so far.
- If an array of objects has been partially constructed when an exception occurs, the destructors for only the array's constructed objects will be called.

Stack unwinding is guaranteed to have been completed when a `catch` handler begins executing. If a destructor invoked due to stack unwinding throws an exception, the program terminates. According to the [isocpp.org FAQ](https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw),³⁸ the choice to terminate is because C++ does not know whether to

- continue processing the exception that led to stack unwinding in the first place or
- process the new exception thrown from the destructor.



Checkpoint

1 (*True/False*) 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)`.

Answer: True.

2 (*True/False*) If an exception occurs during object construction, member objects' and base-class subobjects' destructors will not be called.

Answer: False. If an exception occurs during object construction, destructors will be called for any member objects or base-class subobjects constructed so far. If an array of objects has been partially constructed when an exception occurs, the destructors for only the array's constructed objects will be called.

3 (*True/False*) 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.

Answer: True.

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 the 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 calls `set_new_handler` to specify a function to call when `new` fails. This technique is mainly used in legacy C++ code written before compilers supported throwing `bad_alloc` exceptions.

38. "How Can I Handle a Destructor That Fails?" Accessed April 18, 2023. <https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw>.

When an exception is thrown from the constructor for an object 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

Section 11.5 showed that a `unique_ptr` enables you to ensure that dynamically allocated memory is properly deallocated regardless of whether the `unique_ptr` goes out of scope due to the normal flow of control or an exception. When managing dynamically allocated memory via old-style raw pointers (as might be the case in legacy C++ code), do not allow uncaught exceptions to occur before you release the memory.³⁹ For example, suppose a function contained the following series of statements:

```
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
// ...
```

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 discussed in Chapter 11.

12.8.1 new Throwing bad_alloc on Failure

Figure 12.6 demonstrates `new` throwing `bad_alloc` when `new` fails to allocate memory.⁴⁰ The `for` statement (lines 15–19) inside the `try` block iterates through the array of `unique_ptr` objects named `items` and allocates to each element an array of 500,000,000 `doubles`. Our primary 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. We used a separate test machine with 16 GB of RAM and 256 GB of disk space to produce this program's output and still had to allocate five billion `doubles` to induce a memory allocation failure. 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 21, where the catch handler catches and processes the exception. Lines 22–23 print "Exception occurred:" followed by the message returned from 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 10 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.

39. C++ Core Guidelines, "E.13: Never Throw While Being the Direct Owner of an Object." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-never-throw>.

40. This program's output was produced using Visual C++ on Windows. The executables produced by both `g++` and `clang++` resulted in the message "killed" rather than a `bad_alloc` exception.

```

1 // fig12_06.cpp
2 // Demonstrating standard new throwing bad_alloc when memory
3 // cannot be allocated.
4 #include <array>
5 #include <format>
6 #include <iostream>
7 #include <memory>
8 #include <new> // bad_alloc class is defined here
9
10 int main() {
11     std::array<std::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 = std::make_unique<double[]>(500'000'000);
17             std::cout << std::format(
18                 "items[{}] points to 500,000,000 doubles\n", i++);
19         }
20     }
21     catch (const std::bad_alloc& memoryAllocationException) {
22         std::cerr << std::format("Exception occurred: {}\n",
23             memoryAllocationException.what());
24     }
25 }
```

```

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



You should use the version of new that throws `bad_alloc` exceptions on failure. However, the C++ standard specifies you also can use the version of new that returns `nullptr` upon failure—this version is useful in systems that do not allow exception handling. For this purpose, header `<new>` defines object `std::nothrow` (of type `nothrow_t`), which is used as follows:

```
std::unique_ptr<double[]> ptr{
    new(std::nothrow) double[500'000'000];
```

You cannot use `make_unique` here because it uses the default version of `new`.

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 returns `void`.

The function or lambda is called if `new` fails. 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.

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` (or a derived class of `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 destroyed when either of these functions is called.

Figure 12.7 demonstrates `set_new_handler` (line 21). Function `customNewHandler` (lines 11–14) prints an error message (line 12), then calls `exit` (line 13) to terminate the program. The constant `EXIT_FAILURE` is defined in the header `<cstdlib>`. 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.

```

1 // fig12_07.cpp
2 // Demonstrating set_new_handler.
3 #include <array>
4 #include <cstdlib>
5 #include <format>
6 #include <iostream>
7 #include <memory>
8 #include <new> // set_new_handler is defined here
9
10 // handle memory allocation failure
11 void customNewHandler() {
12     std::cerr << "customNewHandler was called\n";
13     std::exit(EXIT_FAILURE);
14 }
15
16 int main() {
17     std::array<std::unique_ptr<double>[], 1000> items{};
18
19     // specify that customNewHandler should be called on
20     // memory allocation failure
21     std::set_new_handler(customNewHandler);

```

Fig. 12.7 | `set_new_handler` specifying the function to call when `new` fails. (Part I of 2.)

```

22
23     // aim each unique_ptr at a big block of memory
24     for (int i{0}; auto& item : items) {
25         item = std::make_unique<double[]>(500'000'000);
26         std::cout << std::format(
27             "items[{}] points to 500,000,000 doubles\n", i++);
28     }
29 }
```

```

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.)



Checkpoint

1 (*True/False*) When managing dynamically allocated memory via old-style raw pointers (typically in legacy C++ code), it's OK to allow uncaught exceptions to occur before you release the memory.

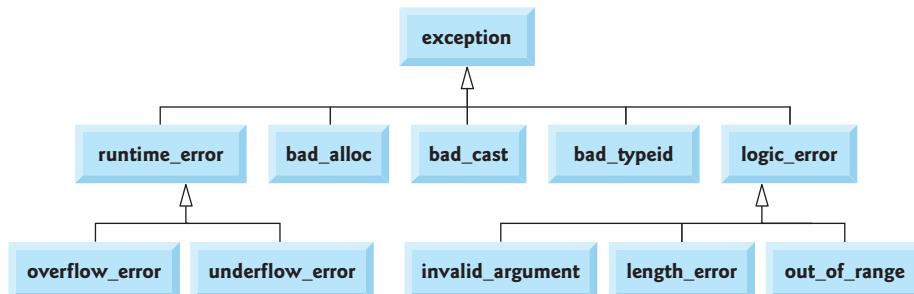
Answer: False. Actually, in this case, you should not allow uncaught exceptions to occur before you release the memory, as this will result in memory leaks.

2 (*True/False*) 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.

Answer: True.

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 C++ standard library exception types—including the new C++20 exception types `nonexistent_local_time`, `ambiguous_local_time` and `format_error`—see

<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.

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`.

A SE

O CG

Base Class exception

The standard library exception hierarchy is headed by base-class `exception` (defined in header `<exception>`). This class contains the `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

O CG

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` (Chapter 20) and
- `bad_typeid` is thrown by `typeid` (Chapter 20).

Derived Classes of runtime_error

Class `runtime_error`, which we used briefly in Sections 12.2 and 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 greater than the largest positive number or less than the largest negative number that can be stored in a given numeric type).
- Class `underflow_error` describes an `arithmetic underflow error`. This is for “subnormal floating-point values.”⁴²

41. C++ Core Guidelines, “E.15: Catch Exceptions from a Hierarchy By Reference.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-ref>.

42. “Subnormal number.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Subnormal_number.

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



Exceptions related by inheritance can be caught by a `catch` specifying a reference to the base-class exception type—you might do this to log an exception (see Section 12.12) before re-throwing it. 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—compilers will issue a warning for this. The base-class `catch` matches all objects of classes derived publicly from that base class, so the derived-class `catch` will never execute.⁴³

Catching All Exceptions



`C++` exceptions need not derive from `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.

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.



Checkpoint

1 (*True/False*) You can use `catch(...)` to catch all exception types thrown in a `try` block, but the type of the caught exception is unknown, and you cannot refer to the exception object inside the exception handler.

Answer: True.

2 (*True/False*) Catching type `exception` is guaranteed to catch all exceptions a program could encounter.

Answer: False. Actually, `C++` exceptions need not derive from class `exception`, so catching type `exception` is not guaranteed to catch all exceptions a program could encounter.

43. `C++ Core Guidelines`, “E.31: Properly Order Your `catch`-Clauses.” Accessed April 18, 2023. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re_catch/.

3 (*True/False*) 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`.

Answer: True

12.10 C++’s Alternative to the `finally` Block: Resource Acquisition Is Initialization (RAII)

In several programming languages 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 release 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 release 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 release the resources automatically.



Checkpoint

1 (*Fill-in*) If you design a class to use RAII, objects of your class will acquire their resources during object construction and release them during _____.

Answer: object destruction.

12.11 Some Libraries 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.⁴⁴ For example,

- the Google C++ Style Guide⁴⁵ indicates that Google does not use exceptions and

44. “C++ Developer Survey ‘Lite’: 2018–02.” February 2018. Accessed April 18, 2023. <https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf>.

45. “Google C++ Style Guide.” Accessed April 18, 2023. <https://google.github.io/styleguide/cppguide.html>.

- the **Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards**⁴⁶ explicitly forbid using `try`, `catch` and `throw`.

When organizations prohibit exceptions, they also prohibit using libraries with functions that might 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 included⁴⁷

- 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 the code by intermixing logic and error handling,
- issues with resource allocation and deallocation and
- the 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, consider the functions in the `<filesystem>` library.⁴⁸ 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.



Checkpoint

1 (*Fill-in*) 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 _____.

Answer: one that sets or returns an error indicator when it encounters a problem.

2 (*True/False*) When organizations prohibit exceptions, they also prohibit using libraries with functions that might throw exceptions—such as the C++ standard library.

Answer: True.

46. “Joint Strike Fighter Air Vehicle C++ Coding Standards,” December 2005. Accessed April 18, 2023. <https://www.stroustrup.com/JSF-AV-rules.pdf>.

47. 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 April 18, 2023. <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>.

48. “Filesystem Library.” Accessed April 18, 2023. <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/doc/html/.
- 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.



Checkpoint

I *(Fill-in)* Logging is built into the C++ standard library.

Answer: False. Actually, logging is not built into the C++ standard library, though you could create your own logging mechanisms using C++’s file-processing capabilities. There are many open-source C++ logging libraries.

12.13 Looking Ahead to Contracts⁴⁹

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 each, show precondition code examples and explain what happens when contracts are violated:

- A **precondition**⁵⁰ 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. Contracts are not yet a standard C++ feature. The syntax we show here could change.

50. “Precondition.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. <https://en.wikipedia.org/wiki/Precondition>.

- A **postcondition**⁵¹ is true after the function successfully returns. Postconditions describe guarantees about 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.

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 non-negative.
- 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.

51. "Postcondition." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. <https://en.wikipedia.org/wiki/Postcondition>.

Design By Contract

Design by contract (DbC)^{52,53,54} 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,

- a function expects client code to meet the function's precondition(s),
- if the preconditions are true, the function guarantees its postcondition(s) will be true, and
- any invariants are maintained.

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.⁵⁵ Another proposal was eventually accepted for inclusion in C++20 but removed⁵⁶ late in C++20’s development cycle due to “lingering design disagreements and concerns.”⁵⁷ So, contracts have been pushed to at least C++23.

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.”⁵⁸

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.⁵⁹



-
52. Bertrand Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1988.
53. Bertrand Meyer, In *Touch of Class: Learning to Program Well with Objects and Contracts*, xvii. Springer Berlin AN, 2016.
54. “Design by Contract.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Design_by_contract.
55. Nathan Meyers, “What Happened to C++20 Contracts?” August 5, 2019. Accessed April 18, 2023. https://www.reddit.com/r/cpp/comments/cmk7ek/what_happened_to_c20_contracts/.
56. Meyers, “What Happened to C++20 Contracts?”
57. Herb Sutter, “Trip Report: Summer ISO C++ Standards Meeting (Cologne),” July 2019. Accessed April 18, 2023. <https://herbsutter.com/2019/07/20/trip-report-summer-iso-c-standards-meeting-cologne/>.
58. Herb Sutter, “Trip Report: Summer Iso C++ Standards Meeting (Rapperswil),” July 2018. Accessed April 18, 2023. <https://herbsutter.com/2018/07/>.
59. Glennen Carnie, “Contract Killing (in Modern C++),” September 18, 2019. Accessed April 18, 2023. <https://blog.feabhas.com/2019/09/contract-killing-in-modern-c/>.



A goal of contracts is to make most functions `noexcept`,⁶⁰ which might enable the compiler to perform additional optimizations. Sutter says, “Gradually switching precondition violations from exceptions to contracts promises to eventually remove a majority of all exceptions thrown by the standard library.”^{61,62}

Contracts Attributes

The contracts proposal⁶³ introduces three attributes of the form

```
[[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 with little runtime overhead compared to the function’s typical execution time. If a level is not specified, the compiler assumes `default`.
- **audit** specifies a contract with **significant runtime overhead** compared to the function’s typical execution time. Such contracts are intended primarily for use during program development.
- **axiom** specifies a contract meant to be enforced by static code checkers rather than at runtime.

60. Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil).”

61. Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil).”

62. 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 1,800 pages—450+ pages cover the language, 1,000+ cover the standard library, and the rest are appendices, bibliography, cross references and indexes. “Throws” appears 422 times—there were 92 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.

63. G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Meyers and B. Stroustrup, “Support for Contract Based Programming in C++,” June 8, 2018. Accessed April 18, 2023. <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html>.



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:

```
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 assert that an integer `exam grade` is in the range 0 through 100, you'd write:

```
[[assert: grade >= 0 && grade <= 100]];
```

The `assert` in the preceding statement is not an `assert` macro.

Early-Access Implementation

There is an early-access contracts implementation in GNU C++⁶⁴ that you can test through the [Compiler Explorer website](#)⁶⁵ (<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

<https://gitlab.com/lock3/gcc-new/-/wikis/contract-assertions>

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 `expects` 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 5–6) a `default` level precondition contract indicating that the `denominator` must not be 0.0. The `default` keyword also can be specified explicitly, as in

```
[[pre default: denominator != 0.0]]
```

64. 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.

65. Copyright © 2012–2019, Compiler Explorer Authors. All rights reserved. Compiler Explorer is by Matt Godbolt. <https://xania.org/MattGodbolt>.

This example can be found at

<https://godbolt.org/z/jn4MK3o9T>

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.

```

1 // fig12_08.cpp
2 // quotient function with a contract precondition.
3 #include <iostream>
4
5 double quotient(double numerator, double denominator)
6   [[pre: denominator != 0.0]];
7
8 int main() {
9     std::cout << "quotient(100, 7): " << quotient(100, 7)
10    << "\nquotient(100, 0): " << quotient(100, 0) << '\n';
11 }
12
13 // perform division
14 double quotient(double numerator, double denominator) {
15     return numerator / denominator;
16 }
```

Fig. 12.8 | quotient function with a contract precondition.

We preset the compiler options for this example to

`-std=c++20 -fcontracts`

which compiles the code with C++20 and experimental contracts support. 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 9 satisfies the precondition and executes successfully, producing:

quotient(100, 7): 14.2857

The `quotient` call at line 10, however, causes a **contract violation**. So, the **default violation handler (handle_contractViolation)** is implicitly called, displays the following error message, then terminates the program:

```
default std::handle_contractViolation called:
./example.cpp 6 quotient denominator != 0.0 default default 0
```

Changing the compilation options to

```
-std=c++2a -fcontracts -fcontract-build-level=off
```

disables contract checking and allows line 10 to execute, producing the output

```
quotient(100, 0): inf
```

Division-by-zero is undefined behavior in C++ but many compilers, including the three key compilers we use throughout this book, return positive infinity (`inf`) or negative infinity (`-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

```
-fcontract-continuation-mode=on
```

Example: Search Function Requiring a Sorted vector

Consider Fig. 12.9, which defines a `binarySearch` function template with two preconditions—the `vector` argument must contain elements and must be sorted. You can test the example at

<https://godbolt.org/z/obMsY5Wo4>

In this example, we defined the function before `main` and placed the preconditions after the parameter list and before the function's body.

```
1 // fig12_09.cpp
2 // binarySearch function with a precondition requiring a sorted vector.
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 template<typename T>
8 int binarySearch(const std::vector<T>& items, const T& key)
9   [[pre: items.size() > 0]]
10  [[pre audit: std::is_sorted(items.begin(), items.end())]] {
11    size_t low{0}; // low index of elements to search
12    size_t high{items.size() - 1}; // high index
13    size_t middle{(low + high + 1) / 2}; // middle element
14    int loc{-1}; // key's index; -1 if not found
15 }
```

Fig. 12.9 | `binarySearch` function with a precondition and a postcondition. (Part I of 2.)

```

16    do { // loop to search for element
17        // if the element is found at the middle
18        if (key == items[middle]) {
19            loc = middle; // loc is the current middle
20        }
21        else if (key < items[middle]) { // middle is too high
22            high = middle - 1; // eliminate the higher half
23        }
24        else { // middle element is too low
25            low = middle + 1; // eliminate the lower half
26        }
27
28        middle = (low + high + 1) / 2; // recalculate the middle
29    } while ((low <= high) && (loc == -1));
30
31    return loc; // return location of key
32}
33
34 int main() {
35     // sorted vector v1 satisfies binarySearch's sorted vector precondition
36     std::vector v1{10, 20, 30, 40, 50, 60, 70, 80, 90};
37     int result1{binarySearch(v1, 70)};
38     std::cout << "70 was " << (result1 != -1 ? "" : "not ")
39         << "found in v1\n";
40
41     // unsorted vector v2 violates binarySearch's sorted vector precondition
42     std::vector v2{60, 70, 80, 90, 10, 20, 30, 40, 50};
43     int result2{binarySearch(v2, 60)};
44     std::cout << "60 was " << (result2 != -1 ? "" : "not ")
45         << "found in v2\n";
46 }
```

Fig. 12.9 | `binarySearch` function with a precondition and a postcondition. (Part 2 of 2.)

The `binarySearch` function template performs a binary search on the `vector`, which requires the `vector` to be in sorted order; otherwise, the result could be incorrect. Our implementation also requires the `vector` to contain elements. So we declared the following preconditions:

```
[[pre: items.size() > 0]]
[[pre audit: is_sorted(begin(items), end(items))]]
```

The first ensures that the `vector` contains elements. The second calls the C++ standard library function `is_sorted` (from header `<algorithm>`⁶⁶) 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 with an efficient 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`.

66. There are over 200 functions in the C++ standard library's `<algorithm>` header. We survey many of these in Chapter 14.

We preset the compiler options for this example to

```
-std=c++2a -fcontracts
```

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 36), and `v2` is unsorted (line 42). With the initial compiler settings, the precondition that ensures the vector is sorted 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

```
-std=c++2a -fcontracts -fcontract-build-level=audit
```

enables audit level contract checking. When the program runs with audit contracts enabled, the vector `v1` in line 37's `binarySearch` call satisfies the precondition in line 10 and, as before, lines 38–39 output

```
70 was found in v1
```

However, the vector `v2` in line 43's `binarySearch` call causes a **contract violation**, resulting in the error message:

```
default std::handle_contractViolation called:
./example.cpp 10 binarySearch<int> is_sorted(begin(items), end(items)) audit
default 0
```

Custom Contract Violation Handler

The examples so far used the default contract violation handler. When a contract violation occurs, a **contract_violation** object is created containing the following information:

- the line number of the violation—returned by member function `line_number`,
- the source-code filename—returned by member function `file_name`,
- the function name in which the violation occurred—returned by member function `function_name`,
- a description of the condition that was violated—returned by member function `comment` and
- the contract level—returned by member function `assertion_level`.

The `contractViolation` is passed to the **violation handler** function of the form

```
void handle_contractViolation(const contractViolation& violation) {
    // handler code
}
```

The experimental contracts implementation in g++ provides a default implementation of this function. If you define your own, g++ will call your version.



Checkpoint

- 1** (*True/False*) 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.

Answer: True.

- 2** (*Fill-in*) The contract attribute _____ specifies a function's preconditions that are checked before the function's body begins executing, and the attribute _____ specifies a function's postconditions that are checked just before the function returns.

Answer: expects, ensures.

- 3** (*Code*) Write an assertion that tests whether an integer `examGrade` is in the range 0 through 100.

Answer: `[[assert: grade >= 0 && grade <= 100]]`;

Note that the `assert` in the preceding statement is not an `assert` macro.

12.14 Wrap-Up

C++ is used to build real-world, mission-critical and business-critical software. The systems it's used for are often massive. In this chapter, you learned that it's essential to eliminate bugs during development and decide how to handle problems once the software is in production.

We discussed 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 exceptions that are 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, the basic exception guarantee, the strong exception guarantee and the 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 without 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 that 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 create more robust code.

Chapter 6 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.

Exercises

- 12.1** If no exceptions are thrown in a `try` block, where does control proceed after the `try` block completes execution?
- 12.2** What happens if an exception is thrown outside a `try` block?
- 12.3** Give a key advantage and a key disadvantage of using `catch(...)`.
- 12.4** What happens if no `catch` handler matches the type of a thrown object?
- 12.5** What happens if several handlers match the type of the thrown object?
- 12.6** Suppose a `catch` handler with a precise match to an exception object type is available. Under what circumstances might a different handler be executed for exception objects of that type?
- 12.7** What does the statement `throw;` in a `catch` handler do?
- 12.8** (*Exceptional Conditions*) List various exceptional conditions that have occurred throughout this textbook. List as many additional exceptional conditions as you can. For each of these exceptions, describe briefly how a program typically would handle the exception using the exception-handling techniques discussed in this chapter. Some typical exceptions are division by zero, arithmetic overflow, array subscript out of bounds, exhaustion of the free store, etc.
- 12.9** (*catch Parameter*) Under what circumstances would you not provide a parameter name when defining the type of object that will be caught by a handler?

12.10 (throw Statement) Where in a program would you normally expect to the following such a statement?

```
throw;
```

What would happen if that statement appeared in a different part of the program?

12.11 (Exception Handling vs. Other Schemes) Compare and contrast exception handling with the various other error-processing schemes discussed in the text.

12.12 (Handling Related Exceptions) Describe a technique for handling related exceptions.

12.13 (Throwing Exceptions from a catch) Suppose a program throws an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself throws the same exception. Does this create infinite recursion? Write a program to check your observation.

12.14 (Catching Derived-Class Exceptions) Write a program using classes from the C++ standard library's exception hierarchy to prove that a `catch` handler specifying a base-class exception type can catch derived-class exceptions.

12.15 (Local-Variable Destructors) Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block.

12.16 (Member-Object Destructors) Write a program illustrating that member-object destructors are called for only those member objects that were constructed before an exception occurred.

12.17 (Catching All Exceptions) Write a program that demonstrates several exception types being caught with the `catch(...)` exception handler.

12.18 (Order of Exception Handlers) Write a program illustrating that the order of exception handlers is important. The first matching handler is the one that executes. Attempt to compile and run your program two different ways to show that two different handlers execute with two different effects.

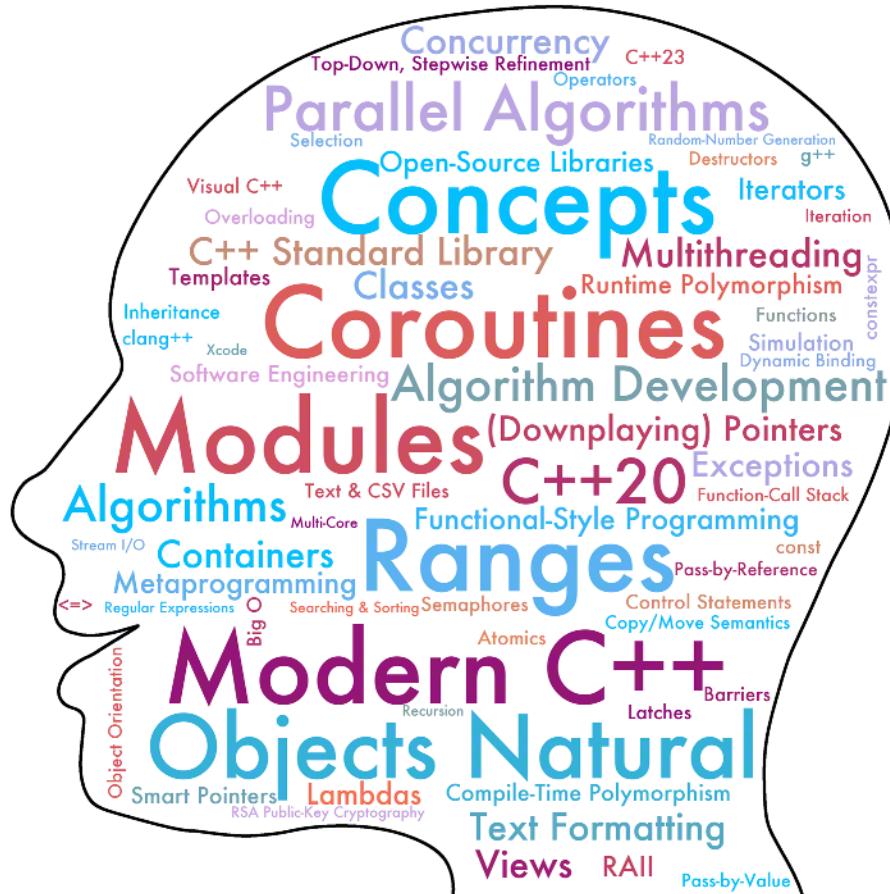
12.19 (Constructors Throwing Exceptions) Write a program that shows a constructor passing information about constructor failure to an exception handler after a `try` block.

12.20 (Rethrowing Exceptions) Write a program that illustrates rethrowing an exception.

12.21 (Uncaught Exceptions) Write a program that illustrates that a function with its own `try` block does not have to catch every possible error generated within the `try`. Some exceptions should slip through to and be handled in outer scopes.

12.22 (Stack Unwinding) Write a program that throws an exception from a deeply nested function and still has the `catch` handler following the `try` block enclosing the initial call in `main` catch the exception.

Data Structures: Standard Library Containers and Iterators



Objectives

In this chapter, you'll:

- Learn more about the C++ standard library's reusable containers, iterators and algorithms.
- 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_iterators` with the `std::copy` and `std::ranges::copy` algorithms to output container elements in a single statement.
- 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.
- Use the `bitset` "near container" to manipulate a collection of bit flags.

Outline

13.1	Introduction	13.8	vector Sequence Container
13.2	A Brief Intro to Big O	13.8.1	Using <code>vectors</code> and Iterators
13.3	A Brief Intro to Hash Tables	13.8.2	<code>vector</code> Element-Manipulation Functions
13.4	Introduction to Containers	13.9	list Sequence Container
13.4.1	Common Nested Types in Sequence and Associative Containers	13.10	deque Sequence Container
13.4.2	Common Container Member and Non-Member Functions	13.11	Associative Containers
13.4.3	Requirements for Container Elements	13.11.1	<code>multiset</code> Associative Container
13.5	Working with Iterators	13.11.2	<code>set</code> Associative Container
13.5.1	Using <code>istream_iterator</code> for Input and <code>ostream_iterator</code> for Output	13.11.3	<code>multimap</code> Associative Container
13.5.2	Iterator Categories	13.11.4	<code>map</code> Associative Container
13.5.3	Container Support for Iterators	13.12	Container Adaptors
13.5.4	Predefined Iterator Type Names	13.12.1	<code>stack</code> Adaptor
13.5.5	Iterator Operators	13.12.2	<code>queue</code> Adaptor
13.6	A Brief Introduction to Algorithms	13.12.3	<code>priority_queue</code> Adaptor
13.7	Sequence Containers	13.13	<code>bitset</code> Near Container
		13.14	Wrap-Up Exercises

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.

Containers, Iterators and Algorithms

This chapter introduces three key standard library components—**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 13.9) and **deque** (pronounced “deck”; Section 13.10).

Iterators

Iterators, which have properties similar to **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 via iterators provides tremendous expressive power when combined with standard library algorithms—sometimes reducing many lines of code to a single statement.

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. https://en.wikipedia.org/wiki/History_of_the_Standard_Template_Library; Accessed April 18, 2023.

Algorithms

Standard library **algorithms** (which we'll cover 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 over 100 algorithms:

- 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,
- 14 in the `<memory>` header's `std` namespace—all 14 also are overloaded in the `std::ranges` namespace for use with C++20 ranges and
- 2 in the `<cstdlib>` header.

For the complete list of algorithms with links to their descriptions, visit:

<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, enabling many algorithms to be applied to containers with significantly different implementations. This also enables you to create new algorithms that can process the elements of multiple container types.

C++20 Ranges

Chapter 6 introduced C++20's **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 with iterators representing its beginning and end can be treated as a C++20 range. In this chapter, we'll use the 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 Templated 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 called the **tail**), and removals are made from the front (also called the **head**).
- **Binary trees** are nonlinear, hierarchical data structures that facilitate **searching** and **sorting** data and **duplicate elimination**.

Each of these data structures has many other interesting applications. We can carefully weave linked objects together with pointers, but 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.

Avoid reinventing the wheel. When possible, use the C++ standard library's preexisting containers, iterators and algorithms.² The prepackaged 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. They were conceived and designed for performance and flexibility.

This chapter begins with two special computer science sections on

- **Big O notation** for expressing how hard algorithms have to work based on the number of items they process, and
- hash tables for high-speed storage and retrieval of data in applications with common real-world characteristics.

13.2 A Brief Intro to Big O

In this section, we'll discuss what Big *O* means and explain the significance of several popular Big *O* notations, including $O(1)$, $O(n)$, $O(\log n)$ and $O(n^2)$. In Chapter 21, we'll explain Big *O* of $O(n \log n)$. On today's desktop computers, which commonly perform billions of operations per second, an algorithm's efficiency, as expressed by its Big *O*, translates to whether a program will run almost instantaneously or take seconds, minutes, hours, days, months or even years to complete. Obviously, you'd prefer algorithms that complete quickly, even though the number of items, *n*, they may be processing is large. This is especially true in today's world of Big Data computing applications. We'll mention Big *O* briefly here, use it extensively in Chapter 14 to characterize the efficiency of standard library algorithms and use it in Chapter 21 to characterize the efficiency of various popular searching and sorting algorithms we'll implement.

***O(1)* Algorithms**

Suppose an algorithm tests 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 1,000 elements, it still requires one comparison. In fact, the algorithm is independent of the number, *n*, of elements in the array. This algorithm is said to have **constant running 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

2. C++ Core Guidelines, "SL.1: Use Libraries Wherever Possible." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rs1-lib>.

has 1,000 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 running time**. $O(n)$ is pronounced “on the order of n ” or simply “order n .”

$O(n^2)$ Algorithms

Now suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array. The first element must be compared with all other elements in the array. The second element must be compared with all elements except the first (it was already compared to the first). The third element must be compared with all elements except the first two. In the end, this algorithm makes $(n - 1) + (n - 2) + \dots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Again, Big O notation highlights the n^2 term, ignoring $n/2$.

Big O is concerned with how an algorithm’s running time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm requires 16 comparisons; with eight elements, 64 comparisons. With this algorithm, **doubling** the number of elements **quadruples** the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm requires eight comparisons; with eight elements, 32 comparisons. Again, **doubling** the number of elements **quadruples** the number of comparisons. Both of these algorithms grow as the square of n , so Big O ignores the constant factor, and both algorithms are considered to be $O(n^2)$, which is referred to as **quadratic running time** and pronounced “on the order of n -squared” or simply “order n -squared.”

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 array’s size doubles, the number of comparisons that the algorithm must perform also doubles. Linear search can provide outstanding performance if the element matching the search key is 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 at or 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 use a more efficient algorithm, such as the C++ standard library algorithm `binary_search`, which we used in Chapter 6 and will use again in Chapter 14. We’ll show how to implement the binary search algorithm in Chapter 21.

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 1,023 elements ($2^{10} - 1$) takes **only 10 comparisons** with binary search. The algorithm compares the search key to the middle

element 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 array's first half from further consideration.
- If it's smaller, we can eliminate the array's second half from further 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 1,023 ($2^{10} - 1$) 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, a much larger 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 running time** and pronounced as “order $\log n$.”

Comparing Big O Notations

The following table lists various common Big O notations and several 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
1,000	1	3	1,000	3,000	10^6
1,000,000	1	6	1,000,000	6,000,000	10^{12}

In the implementation of the standard library containers (Chapter 13) and algorithms (Chapter 14), 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.

13.3 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:

<https://www.ssa.gov/employer/randomization.html>

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. This is impractical for most applications that use Social Security numbers as keys.

Numerous applications have this problem—namely, that either the keys are of the wrong type (e.g., not positive integers usable as array subscripts) or they’re of the right type, but sparsely spread over a huge range, such as Social Security numbers for a small company’s employees. 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 a data item, the scheme can convert the application’s key rapidly into an array index (a process called **hashing**), and the item 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, it simply applies the conversion to the key (again, called **hashing**), producing 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**, which occurs when two different keys “hash into” the same array element (or cell). 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 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 hash table’s number of occupied to its total number of cells. The closer this ratio gets to 1.0, the greater the chance of collisions, which slow data insertion and retrieval.



A hash table's load factor 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 associative containers **`unordered_set`**, **`unordered_multiset`**, **`unordered_map`** and **`unordered_multimap`**, which we'll study in this chapter, 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 software reuse.

Another interesting application of hashing is with virtual memory³ operating systems. Programs and data are maintained on massive secondary storage and brought into more limited primary memory (and even more limited and faster cache memory) to execute. The trick to running applications efficiently is to keep in primary memory (and cache memory) the portion of the program that needs to execute at a given time. Finding those program pieces in virtual memory involves searching the very large data structures that define it. Hashing is often used for that purpose.

13.4 Introduction to Containers⁴

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 many live-code examples in the following sections.

Sequence Containers

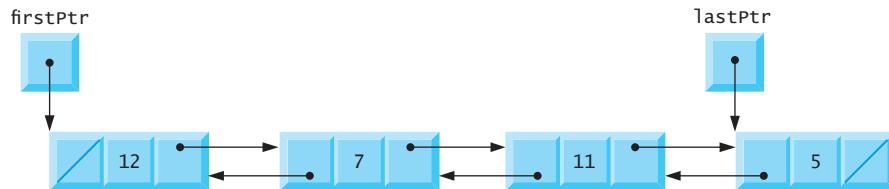
The **sequence containers** represent linear data structures with all of their elements conceptually "lined up in a row," such as **arrays**, **vectors** and linked lists, all of which are sortable. The five **sequence containers** are

- **array** (Chapter 6)—Fixed size. Elements are contiguous in memory. Direct access (also called random access) to any element.
- **vector** (Section 13.8)—Resizable. Elements are contiguous in memory. Rapid insertions and deletions at the back. Direct access to any element.
- **list** (Section 13.9)—Resizable doubly linked list, rapid insertion and deletion anywhere. A **doubly linked list** supports iterating forward or backward through the list and is often implemented with two "start pointers"—one points to the list's first element and one to the last. Each node contains pointers to the previous

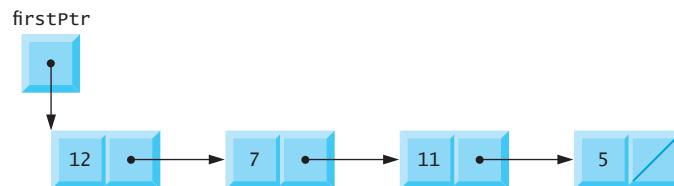
3. "Virtual Memory." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Virtual_memory.

4. 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.

node and next node (hence the term doubly linked list), as illustrated by the horizontal arrows in the following diagram. The slashes (/) in the first and last nodes represent `nullptrs`:



- **`deque`** (Section 13.10)—Resizable. Rapid insertions and deletions at the front or back. Direct access to any element.
- **`forward_list`**—Resizable singly linked list, rapid insertion and deletion anywhere. A **singly linked list** typically maintains a pointer to its first node, and each node contains one pointer to the next node (hence the term singly linked list), as illustrated by the horizontal arrows in the following diagram. The slash (/) in the last node represents a `nullptr`:



Linked-list containers do not support random 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 (usually implemented as binary trees^{5,6}) that typically can quickly locate elements. 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

- **`multiset`** (Section 13.11.1)—Rapid lookup, duplicates allowed.
- **`set`** (Section 13.11.2)—Rapid lookup, no duplicates allowed.
- **`multimap`** (Section 13.11.3)—Rapid key-based lookup, duplicate keys allowed. One-to-many mapping.
- **`map`** (Section 13.11.4)—Rapid key-based lookup, no duplicate keys allowed. One-to-one mapping.

5. “`set`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/set>.

6. “`map`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/map>.

The four **unordered associative containers** (implemented using hashing^{7,8}) are

- **`unordered_multiset`**—Rapid lookup, duplicates allowed.
- **`unordered_set`**—Rapid lookup, no duplicates allowed.
- **`unordered_multimap`**—One-to-many mapping, duplicates allowed, rapid key-based lookup.
- **`unordered_map`**—One-to-one mapping, no duplicates allowed, rapid key-based lookup.

Container Adaptors

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 13.13) for maintaining sets of flag values, **strings** and **valarrays** for performing high-speed mathematical vector operations⁹ (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**.

13.4.1 Common Nested Types in Sequence and Associative Containers

The table below shows the common types defined inside each sequence and associative container's class definition. These **nested types** are used in template-based variable declarations, function parameters and function return types, as you'll see in this chapter and Chapter 14. For example, each container's **`value_type`** always represents the container's element type.

7. “`unordered_set`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/container/unordered_set.

8. “`unordered_map`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/container/unordered_map.

9. 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/>.

Nested type	Description
<code>allocator_type</code>	The type of 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. Custom allocators are beyond this book's scope.
<code>value_type</code>	The type of the container's elements.
<code>reference</code>	The type used to declare a reference to a container element.
<code>const_reference</code>	The type used to declare a reference to a <code>const</code> container element.
<code>pointer</code>	The type used to declare a pointer to a container element.
<code>const_pointer</code>	The type used to declare a pointer to a <code>const</code> container element.
<code>iterator</code>	An iterator that points to a container element.
<code>const_iterator</code>	An iterator that points to an element of a <code>const</code> container. Used only to read elements and to perform <code>const</code> operations.
<code>reverse_iterator</code>	A reverse iterator that points to a container element. Iterates through a container back-to-front . Not provided for <code>forward_list</code> .
<code>const_reverse_iterator</code>	A reverse iterator that points to a container element and can be used only to read elements and to perform <code>const</code> operations. Used to iterate through a container in reverse. Not provided for <code>forward_list</code> .
<code>difference_type</code>	A signed 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 iterator's overloaded minus (-) operator and by the function <code>std::distance</code> .
<code>size_type</code>	The unsigned type used to count items in a container and index through a random-access sequence container.

13.4.2 Common Container Member and Non-Member Functions

Most containers provide similar functionality. Many operations apply to all containers, and others apply to specific container categories. The following tables describe the commonly available functions in most 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 [cppreference.com member function table](#).¹⁰ Several member functions we do not list in this section are covered throughout the chapter as we present various sequence containers, associative containers and container adaptors.

Container Special Member Functions

The following table describes the 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`.

10. "Container Library—Member Functions Table." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container>.

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.
<code>copy operator=</code>	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.
<code>move operator=</code>	Moves the contents 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 to destroy the container elements.

Non-Member Relational and Equality Operators

The `<`, `<=`, `>`, `>=`, `==` and `!=` operators are supported by most containers. In C++17 and earlier, these are overloaded as non-member functions. In C++20, the `<`, `<=`, `>`, `>=` and `!=` operators are synthesized by the compiler using the new **three-way comparison operator** `<=>` and the `==` 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`.

Member Functions That Return Iterators

The following table shows container member functions that return iterators. The container `forward_list` does not 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>	Returns a <code>const_iterator</code> referring to the container's first element .
<code>cend</code>	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 .

Container member function (Cont.)	Description
<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>	Returns a <code>const_reverse_iterator</code> referring to the container's last element .
<code>crend</code>	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 which do or do not support it. For a container's complete list of member functions, see its documentation page, which you can access from

<https://en.cppreference.com/w/cpp/container>

Container member function	Description
<code>clear</code>	Removes all elements from the container. Not supported for <code>arrays</code> .
<code>contains</code>	Returns <code>true</code> if the specified key is present in the container; otherwise, returns <code>false</code> . Supported only for associative containers.
<code>empty</code>	Returns <code>true</code> if the container is empty; otherwise, returns <code>false</code> .
<code>emplace</code>	Constructs an item in place where it will reside in the container. If there is already an item in that position, it constructs the object in a separate location, then moves it into place with move assignment. Not supported for <code>arrays</code> . A <code>forward_list</code> supports <code>emplace_after</code> and <code>emplace_front</code> , not <code>erase</code> . The associative containers provide <code>emplace</code> and <code>emplace_hint</code> .
<code>erase</code>	Removes one or more elements from the container. Not supported for <code>arrays</code> . A <code>forward_list</code> supports <code>erase_after</code> , not <code>erase</code> .
<code>extract</code>	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> .
<code>insert</code>	Inserts an item in the container. This is overloaded to support copy and move semantics . Not supported for <code>arrays</code> . A <code>forward_list</code> supports <code>insert_after</code> , not <code>insert</code> .
<code>max_size</code>	Returns the maximum number of elements for a container.
<code>size</code>	Returns the number of elements currently in the container. Not supported for <code>forward_lists</code> .
<code>swap</code>	Swaps the contents of two containers.

13.4.3 Requirements for Container Elements

Before using a standard library container, it's important to ensure that the element type 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

https://en.cppreference.com/w/cpp/named_req

In C++20, many named requirements are formalized as **concepts**, which we'll discuss in Chapters 14 and 15.

13.5 Working with Iterators

Iterators have many similarities to pointers. They point to elements in sequence containers and associative containers, but they also hold state information sensitive to the particular containers on which they operate. So, iterators are implemented for each type of container. Iterator operation syntax is uniform across containers. For example, applying `*` to an iterator dereferences it so that you can access the referenced element. Applying `++` to an iterator moves it to the container's next element. This uniformity is a key aspect of iterators, enabling standard library algorithms to operate on various container types using compile-time polymorphism.

Sequence containers and associative containers provide member functions `begin` and `end`. Function `begin` returns an iterator pointing to the container's first element. Function `end` returns an iterator pointing to the **first element past the end of the container** (one past the end)—a nonexistent element that's frequently used to determine when the end of a container is reached. You'll often use this in equality or inequality comparisons to determine whether an incremented iterator has reached the end of the container.

13.5.1 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.

```

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
```

Fig. 13.1 | Demonstrating input and output with iterators.

istream_iterator

Line 10 creates an `istream_iterator` 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 initializes `number1` with the value. The dereferencing operator `*` applied to `inputInt` gets the value from the stream. Line 13 positions `inputInt` to the next value in the input stream. Line 14 inputs the next `int` from `inputInt` and initializes `number2` with it. Either prefix or postfix increment can be used. We use the prefix form for performance reasons because it does not create a temporary object.

ostream_iterator

Line 17 creates an `ostream_iterator` capable of **inserting** (outputting) `int` values in the standard output object `cout`. Line 20 outputs an integer to `cout` by assigning to the dereferenced iterator (`*outputInt`) the sum of `number1` and `number2`.

13.5.2 Iterator Categories

The following table describes the iterator categories. Each provides a specific set of functionality. Throughout this chapter, we discuss which iterator category each container supports. In Chapter 14, you'll see that an algorithm's minimum iterator requirements determine which containers can be used with that algorithm.

Iterator category	Description
input	Used to read an element from a container. 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. 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.
forward	Has the capabilities of input and output iterators and retains its position in the container. Such iterators can 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—that is, to jump forward or backward by an arbitrary number of elements. These can also be compared with relational operators.
contiguous	A random-access iterator that requires elements to be stored in contiguous memory locations.

13.5.3 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.** Pointers into built-in arrays can be used as iterators. 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		Ordered associative containers	
vector	contiguous	set	bidirectional
array	contiguous	multiset	bidirectional
deque	random access	map	bidirectional
list	bidirectional	multimap	bidirectional
forward_list	forward		

Container	Iterator type	Container	Iterator type
Unordered associative containers		Container adaptors	
<code>unordered_set</code>	forward	<code>stack</code>	none
<code>unordered_multiset</code>	forward	<code>queue</code>	none
<code>unordered_map</code>	forward	<code>priority_queue</code>	none
<code>unordered_multimap</code>	forward		

13.5.4 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 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 container elements. Using `const_iterators` where appropriate is another example of the principle of least privilege.

13.5.5 Iterator Operators

The following table shows operators for each iterator type. In addition to the operators shown for all iterators, iterators must provide default constructors (forward iterators and higher), copy constructors and copy assignment operators. A forward iterator supports ++ and all 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.

Iterator operation	Description
All iterators	
<code>++p</code>	Preincrement an iterator, moving it to the next element.
<code>p++</code>	Postincrement an iterator, moving it to the next element.
<code>p = p1</code>	Assign one iterator to another.

Iterator operation	Description
Input iterators	
<code>*p</code>	Dereference an iterator as a <code>const lvalue</code> .
<code>p->m</code>	Use the iterator to access 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 <code>lvalue</code> .
<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>	Pred decrement an iterator, moving to the previous element.
<code>p--</code>	Post decrement an iterator, moving to the previous element.
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> .

13.6 A Brief Introduction to Algorithms

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**—known as **half-open ranges**. Many now 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.

13.7 Sequence Containers

The C++ standard 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.

Performance and Choosing the Appropriate Container

Section 13.4.2 presented the operations common to most standard library containers. Beyond these operations, each container typically provides various other capabilities. Many are common to several containers, but they're not always equally efficient. The C++ Core Guidelines say `vector` is **satisfactory for most applications**.¹¹ However, you should profile your application to ensure the container you choose is the correct one for your use-case and performance requirements.

Insertion at the back of a `vector` is efficient. The `vector` grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element at the front or in the middle of a `vector`. Every item after the insertion (or deletion) point must be moved because `vector` elements **occupy contiguous cells in memory**.

For applications that require frequent insertions and deletions at both ends of a container, use a `deque` rather than a `vector`. A `deque` does not move any elements for insertions and deletions at the front,¹² making it more efficient than `vector`.¹³

Applications with frequent insertions and deletions in the middle and/or at the extremes of a container sometimes use a `list` due to its efficient implementation of insertion and deletion anywhere in the data structure.

13.8 `vector` Sequence Container

The `vector` container (introduced in Section 6.15) is a dynamic data structure with contiguous memory locations. It provides rapid indexed access with the overloaded subscript operator `[]`. Choose the `vector` container for the best random-access performance in a container that can grow. When a `vector`'s capacity is exceeded, it

- allocates a larger built-in array,

11. C++ Core Guidelines, “SL.con.2: Prefer Using STL `vector` By Default Unless You Have a Reason to Use a Different Container.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-vector>.

12. “`std::deque`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/deque>.

13. “`std::vector`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/vector>.

- 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.

13.8.1 Using vectors and Iterators

Figure 13.2 illustrates several vector member functions, many of which are available in every sequence container and associative container. As we add items to a `vector<int>` in this example, we call our `showResult` function (lines 9–12) to display the new value as well as the vector's

- `size`—the number of elements it currently contains and
- `capacity`—the number of elements it can store before it needs to dynamically resize itself to accommodate more elements.

```

1 // fig13_02.cpp
2 // Standard Library vector class template.
3 #include <format>
4 #include <iostream>
5 #include <ranges>
6 #include <vector> // vector class-template definition
7
8 // display value appended to vector and updated vector size and capacity
9 void showResult(int value, size_t size, size_t capacity) {
10     std::cout << std::format("appended: {}; size: {}; capacity: {}\\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 `integers`. `vector`'s default constructor creates an empty vector with its size and capacity set to 0. So, the vector will have to allocate memory when elements are added to it. Lines 17–18 display the `size` and `capacity`. Function `size` returns the number of elements currently stored in the container.¹⁴ Function `capacity` returns the vector's current capacity.

```

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
```

14. `forward_list` does not have the `size` member function.

`vector` Member Function `push_back`

Lines 21–24 call `push_back` to append one element at a time to the `vector`. Each loop iteration calls `showResult` to display the added item and the `vector`'s new `size` and `capacity`. Function `push_back` 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` when you add a new element, the `vector` increases its capacity to accommodate more elements.

```
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
```

Updated size and capacity After Modifying a `vector`

The C++ standard does not specify how a `vector` grows to accommodate more elements—a time-consuming operation. 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 in g++

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:

```

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

```

Other vector Sizing Considerations

 Some programmers allocate a large initial capacity. If a `vector` stores a small number of elements, such capacity may waste 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 time required to perform various `vector` operations.**

 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 `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 `constIterator` using `vector` member function `cbegin`, which returns a `const_iterator` to the `vector`'s first element. We use `auto` to infer the control variable's type

```
vector<int>::const_iterator
```

which simplifies the code and reduces errors when working with more complex types.

```

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 10
```

The loop continues as long as `constIterator` has not reached the `vector`'s end. This is determined by comparing `constIterator` to the result of calling the `vector`'s `cend` function. When `constIterator` equals this value, the loop terminates. **Attempting to dereference an iterator positioned outside its container is a logic error.** The iterator returned by `end` or `cend` should never be dereferenced or incremented.

Line 30 dereferences `constIterator` to get the current element's value. Remember that the iterator acts like a pointer to an element, and operator `*` is overloaded to return a reference to the element. The expression `++constIterator` (line 29) positions the iterator to the `vector`'s next element. You could replace this loop with the following more straightforward range-based `for` statement:

```
for (auto const& item : integers) {  
    std::cout << item << ' ';  
}
```

The range-based `for` uses iterators “under the hood.”

Displaying the vector's Contents in Reverse with `const_reverse_iterators`

Lines 36–39 iterate through the `vector` in reverse. The `vector` member functions `crbegin` and `crend` each return `const_reverse_iterators` representing the starting and ending points when iterating through a container in reverse. Most sequence containers and the ordered associative containers support reverse iteration. Class `vector` also provides member functions `rbegin` and `rend` to obtain `non-const reverse_iterators`.

```
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.crbegin()};  
37         reverseIterator != integers.crend(); ++reverseIterator) {  
38         std::cout << *reverseIterator << ' ';  
39     }  
40  
41     std::cout << "\n";  
42 }
```

```
Output integers in reverse using iterators: 10 9 8 7 6 5 4 3 2 1
```

`shrink_to_fit`

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.

13.8.2 vector Element-Manipulation Functions

Figure 13.3 illustrates functions for retrieving and manipulating `vector` elements. Line 12 initializes a `vector<int>` with a braced initializer. In this case, we declared the `vector`'s type as `std::vector`. The compiler uses class template argument deduction (CTAD) to infer the element type from the initializer list's `int` values. Line 13 uses a `vector` constructor that takes two iterators as arguments to initialize `integers` with a copy of the elements in the range `values.cbegin()` up to, but not including, `values.cend()`.

```

1 // fig13_03.cpp
2 // Testing standard library vector class template
3 // element-manipulation functions.
4 #include <algorithm> // copy algorithm
5 #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

```

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`. The constructor's first argument specifies the output stream, and the second argument is a string specifying the separator for the values output—in this case, a space character. We use the `ostream_iterator` (header `<iterator>`) to output the `vector` contents in this example.

copy Algorithm

Line 17 uses standard library algorithm `copy` (header `<algorithm>`) to output the 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 `integers.cbegin()` up to, but not including, `integers.cend()`. These must satisfy `input iterator` requirements, such as `const_iterators`, enabling values to be read from a container. They must also refer to the same container such that applying `++` to the first iterator repeatedly causes it to reach the second iterator argument eventually. The third argument specifies where to copy the elements. This must be an `output iterator` through which a value can be stored or output. The output iterator is an `ostream_iterator` attached to `cout`, so line 17 copies the elements to the standard output.

```

16     std::cout << "integers contains: ";
17     std::copy(integers.cbegin(), integers.cend(), output);
18

```

```
integers contains: 1 2 3 4 5
```

Common Ranges

Before C++20, a “range” of container elements was described by iterators specifying the starting position and the one-past-the-end position, typically via calls to a container’s `begin` and `end` (or similar) member functions. As of C++20, the C++ standard refers to

such ranges as **common ranges**, so they're not confused with C++20 ranges. From this point forward, we'll use 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`

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 an iterator pointing to the vector's first element. Similarly, function `back` returns a reference to the vector's last element, whereas `end` returns an iterator pointing to the location after the last element. The results of `front` and `back` are undefined when called on an empty vector.

```
19     std::cout << std::format("\nfront: {}\nback: {}\\n\\n",
20         integers.front(), integers.back());
21
```

```
front: 1
back: 5
```

Accessing vector Elements

Lines 22–23 illustrate two ways to access `vector` (or `deque`) elements. Line 22 uses the `[]` operator, which returns 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, the `at` function throws an `out_of_range` exception, as demonstrated in Section 6.15.

```
22     integers[0] = 7; // set first element to 7
23     integers.at(2) = 10; // set element at position 2 to 10
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 second element, so 22 becomes the second element, and the original second element is now the third. Other versions of `insert` allow

- inserting multiple copies of the same value starting at a given position or
- inserting a range of values from another container, starting at a given position.

The version of `insert` in line 26 returns an iterator pointing to the inserted item.

```

25 // insert 22 as second 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

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 **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 representing a **range of elements** with **input iterators** representing its beginning and end—in this case, a **vector**. Most pre-C++20 algorithms in the **<algorithm>** header now have C++20 ranges versions 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 C++20 ranges, a **sentinel** is an object that represents when the end of the container has been reached. Chapter 14 demonstrates many C++20 ranges algorithms.

vector Member Function erase

Lines 31 and 36 use two **erase** member functions that are available in most sequence 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 erases elements in the common range specified by the two iterator arguments—in this case, all the elements. Line 38 uses the member function **empty** (defined for all containers and adaptors) to confirm that the **vector** is empty.

```

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 << std::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

Usually, **erase** destroys the objects it erases. However, erasing an element that is a pointer to a dynamically allocated object does not **delete** the object it points to, potentially causing a memory leak. Again, this is why you should not manage dynamically allocated memory with raw pointers. A **unique_ptr** (Section 11.5) element would release the dynamically allocated memory. If the element is a **shared_ptr** (Chapter 20), 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 its second and third arguments to specify a common range of elements to insert into the **vector** (in this case, from **values**). Remember that the ending location specifies the position in the sequence **after** the last element to insert; copying occurs up to, but not including, this location. This version of **insert** returns an iterator pointing to the first item that was inserted. If nothing was inserted, the function returns its first argument.

```
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

vector Member Function clear

Finally, line 46 uses the member function **clear** to empty the **vector**. This does not reduce the **vector**'s capacity. Other than array, which is fixed in size, all sequence containers and associative containers provide **clear**.

```
45     // empty integers; clear empties a collection
46     integers.clear();
47     std::cout << std::format("\nAfter clear, integers {} empty\n",
48                           integers.empty() ? "is" : "is not");
49 }
```

After clear, integers is empty

13.9 List Sequence Container

The **List** sequence container (from header `<list>`) allows insertion and deletion operations at any location in the container. The **list** container is implemented as a **doubly linked list**¹⁵—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. If most of the insertions and deletions occur at the ends of the container, consider using **deque** (Section 13.10) or **vector**. Recall that **deque**'s implementation of inserting at the front is more efficient than **vector**'s.



15. “`std::list`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/list>.

forward_list Container

The **forward_list** sequence container (header `<forward_list>`) 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 common sequence container member functions discussed in Section 13.4, `list` provides member functions `splice`, `push_front`, `pop_front`, `emplace_front`, `emplace_back`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`. Several of these are `list`-optimized implementations of standard library algorithms we show in Chapter 14. Both `push_front` and `pop_front` are also supported by `forward_list` and `deque`. Figure 13.4 demonstrates several `list` features. Many of the functions presented in Figs. 13.2–13.3 also can be used with `lists`. We focus on the new features in this example’s discussion.

```

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

```

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 its 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`.

```
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
```

`List` Member Function `sort`

Line 34 uses `list` member function `sort` to arrange the elements in ascending order. A second version of `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 are sorted. This version is useful for sorting a `list` in descending order or customizing how elements are compared to determine the sort order.

```
34     values.sort(); // sort values  
35     std::cout << "\nvalues after sorting contains: ";  
36     printList(values);  
37 }
```

```
values after sorting contains: 1 2 3 4
```

`List` Member Function `splice`

Line 46 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 common range to remove from the `list` in the second argument and insert at the location specified in the first argument. A `forward_list` provides a similar member function named `splice_after`.

```

38 // insert elements of ints into otherValues
39 std::vector<int> ints{2, 6, 4, 8};
40 std::list<int> otherValues{}; // create list of ints
41 otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
42 std::cout << "\nAfter insert, otherValues contains: ";
43 printList(otherValues);
44
45 // remove otherValues elements and insert at end of values
46 values.splice(values.cend(), otherValues);
47 std::cout << "\nAfter splice, values contains: ";
48 printList(values);
49

```

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`, returning `true` if the predicate's first argument should be placed before the second.

```

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

List Member Function pop_front and pop_back

Line 67 uses **pop_front** to remove the **list**'s first element, and line 68 uses **pop_back** to remove the last element. This function is available for **sequence containers**, except **array** and **forward_list**.

```

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

List Member Function unique

Line 72 uses **list** function **unique** to remove **duplicate adjacent elements**. If the **list** is sorted, 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.

```

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**.

```

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

List Member Functions assign and remove

Line 83 uses **list** member function **assign** (available in all **sequence containers**) to replace **values**' contents with a common range of elements from **otherValues**. A second version of **assign** replaces the **list**'s contents with the specified number of copies of the

value specified in its second argument. Line 92 uses `list` member function `remove` to delete all copies of the value 4 from the `list`.

```

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
```

13.10 deque Sequence Container

Class `deque` (header `<deque>`)—short for “double-ended queue”—provides many benefits of vectors and lists in one container. It’s implemented to provide efficient indexed access (using subscripting) for reading and modifying elements, like a `vector` or array. It also provides efficient insertion and deletion at its front and back, like a `list`. Class `deque` supports random-access iterators, so deques can be used with all standard library algorithms. A common use of a deque is to maintain a first-in, first-out queue of elements. In fact, a deque is the default container for the queue adaptor¹⁶ (Section 13.12.2).

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.¹⁷ Due to a deque’s noncontiguous memory layout, its iterators must be more “intelligent” than those for vectors, arrays or built-in arrays. A deque provides the same 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.

Figure 13.5 demonstrates several deque features. Many functions presented in Figs. 13.2–Fig. 13.4 also can be used with deque.

16. “`std::queue`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/queue>.

17. This is an implementation-specific detail, not a requirement of the C++ standard.

```
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 <iostream> // 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
```

Fig. 13.5 | Standard library deque class template.

Line 9 instantiates a deque that can store `double` values, then lines 13–15 use member functions `push_front` and `push_back` to insert elements at its beginning and end.

Lines 20–22 use the subscript operator to retrieve each element's value for output. The loop condition uses member function `size` to ensure that we do not attempt to access an element outside the deque's bounds. 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 `pop_front` to remove the deque's first element. Line 29 uses the subscript operator to obtain an *lvalue*, which we use to assign a new value to element 1 of the deque.

13.11 Associative Containers

The associative containers provide direct access to store and retrieve elements via **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 most of the same capabilities as their ordered counterparts. The primary difference between ordered and unordered associative containers is that unordered ones do not maintain their keys in sorted order. If your keys are not comparable, you must use the unordered containers. This section focuses on the ordered associative containers.

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.¹⁸ However, this requires the keys to be hashable. For hashable-type requirements, see the documentation for `std::hash`.¹⁹ For an introduction to hashing, see Section 13.3.

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 multiple values for the same key and a **`map`** allows only one value per key. In addition to the common container member functions, ordered associative containers support several member functions specific to associative containers. You can combine the contents of associative containers of the same type using the **`merge`** function. Examples of the ordered associative containers and their common member functions are presented in the next several subsections.

13.11.1 `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. Section 14.5 discusses function objects in detail. Here, 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 comparison operators. A **`multiset`** sup-

18. “Containers Library.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container>.

19. “`std::hash`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/utility/hash>.

ports **bidirectional iterators**. If the order of the keys is not important, consider **unordered_multiset** (header <unordered_set>), but keep in mind that it supports only forward iterators.

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 13.11.2) provide the same basic functionality. Line 12 creates the **multiset**, using the function object **std::less<int>** to specify the keys' sort order. The compiler knows this **multiset** contains **int** values, so you can specify **std::less<int>** simply as **std::less<>**—the compiler will infer that **std::less** compares **int** values. Also, **std::less<>** is the default for a **multiset**, so line 12 can be simplified as

```
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 <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

```

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**.

```

13     std::cout << std::format("15s in ints: {}", ints.count(15));
14

```

```
15s in ints: 0
```

multiset Member Function insert

Lines 16–17 use one of the several overloaded versions of the **insert** member function 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 that specify a common range to add to the **multiset** from another container. For several other overloads, see

<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 << std::format("15s in ints: {}\\n\\n", ints.count(15));
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 first locations of the values 15 and 20 in the `multiset`. The range-based `for` loop iterates through each item in `{15, 20}`, which creates an `initializer_list`. 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. Usually, you'd use `find` if you need to use the iterator that points to the found element. Here, we could have used the `count` member function—if it returns 0, the item is not in the `multiset`.

```

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 << std::format("Found {} in ints\\n", i);
24        }
25        else {
26            std::cout << std::format("Did not find {} in ints\\n", i);
27        }
28    }
29

```

```

Found 15 in ints
Did not find 20 in ints

```

multiset Member Function `contains` (C++20)

Lines 31–38 use the C++20 member function `contains` (line 32) 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}`.

```

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 << std::format("Found {} in ints\\n", i);
34        }
35        else {
36            std::cout << std::format("Did not find {} in ints\\n", i);
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

```
std::ostream_iterator<int>{std::cout, " "}
```

This creates a temporary `ostream_iterator` and immediately passes it to `copy`. We chose this approach because the `ostream_iterator` is used only once in this example.

```
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
```

`multiset` Member Functions `lower_bound` and `upper_bound`

Line 49 uses functions `lower_bound` and `upper_bound`, available in all **ordered 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 they return the end iterator 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.

```
46 // determine lower and upper bound of 22 in ints  
47 std::cout << std::format(  
48 " \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
```

`pair` Objects and `multiset` Member Function `equal_range`

The `multiset` function `equal_range` returns a `pair` containing the results of calling both `lower_bound` and `upper_bound`. A `pair` associates two values. Line 52 creates and initializes a `pair` object called `p`. We use `auto` to infer the variable's type from its initializer. The `pair` returned by `equal_range`, which will contain two `iterators` or `const_iterators`, depending on whether the `multiset` is `const`. A `pair` contains two `public` data members called `first` and `second`—their types depend on the `pair`'s initializers.

```

51 // use equal_range to determine lower and upper bound of 22 in ints
52 auto p{ints.equal_range(22)};
53 std::cout << std::format(
54     "lower_bound(22): {}\\nupper_bound(22): {}\\n",
55     *(p.first), *(p.second));
56 }

```

```

lower_bound(22): 22
upper_bound(22): 30

```

Line 52 uses `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 them.

Heterogeneous Lookup

Before 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**.

13.11.2 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**. When a duplicate is inserted, it is ignored. This is the intended mathematical behavior of a `set`, so it's not considered an error. A `set` supports **bidirectional iterators**. If the order of the keys is not important, consider **unordered_set** (header `<unordered_set>`), but keep in mind that it supports only forward iterators, and its keys must be hashable.

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 element type. Line 10 is equivalent to

```
std::set<double, std::less<double>> doubles{2.1, 4.2, 9.5, 2.1, 3.7};
```

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`. Again, a `set` does not allow duplicates.

```
1 // fig13_07.cpp
2 // Standard library set class template.
3 #include <algorithm>
4 #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
```

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.

```
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 << std::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
```

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.

```

25  // insert 9.5 in doubles
26  p = doubles.insert(9.5); // value already in set
27  std::cout << std::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

```

13.11.3 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. 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>**.

In a **multimap**, 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**.

Creating a multimap Containing Key–Value Pairs

Figure 13.8 demonstrates the **multimap** associative container (header `<map>`). 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

```
std::multimap<int, double, std::less<int>> pairs{};
```

```

1 // fig13_08.cpp
2 // Standard Library multimap class template.
3 #include <iostream>
4 #include <map> // multimap class-template definition
5
6 int main() {
7     std::multimap<int, double> pairs{}; // create multimap
8

```

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.

```

9   std::cout << std::format("Number of 15 keys in pairs: {}\n",
10  pairs.count(15));
11

```

Number of 15 keys in pairs: 0

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, inferring the types of its arguments. In this case, `first` represents a key (15) of type `int`, and `second` represents a value (99.3) of type `double`. 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.

```

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 << std::format("Number of 15 keys in pairs: {}\n\n",
16     pairs.count(15));
17

```

Number of 15 keys in pairs: 2

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});

```

Lines 19–23 insert five additional pairs into the `multimap`. Lines 28–30 output the 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.

```

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 << std::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
```

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 creating 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:

```
std::multimap<int, double> pairs{
    {10, 22.22}, {20, 9.345}, {5, 77.54}};
```

13.11.4 map Associative Container

The `map` associative container (from header `<map>`) performs fast storage and retrieval of unique keys and associated values. A single value can be associated with each unique 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`. If the order of the keys is not important, consider an `unordered_map` (header `<unordered_map>`), but keep in mind that the keys must be hashable.

Figure 13.9 demonstrates a `map` (lines 9–10). 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 the 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.

```
1 // fig13_09.cpp
2 // Standard library class map class template.
3 #include <iostream>
4 #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 }
```

Fig. 13.9 | Standard library `map` class template. (Part 1 of 2.)

```

12 // walk through elements of pairs
13 std::cout << "pairs contains:\nKey\tValue\n";
14 for (const auto& pair : pairs) {
15     std::cout << std::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 // walk through elements of pairs
22 std::cout << "\nAfter updates, pairs contains:\nKey\tValue\n";
23 for (const auto& pair : pairs) {
24     std::cout << std::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
```

Fig. 13.9 | Standard library `map` class template. (Part 2 of 2.)

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`.

13.12 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.

13.12.1 stack Adaptor

Class **stack** (from header `<stack>`) enables insertions into and deletions from the underlying container at one end (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**.²⁰ The **stack** operations are

- **push** to insert an element at the **stack**'s **top**—implemented by calling the underlying container's **push_back** member function,
- **emplace** to construct an element in place at the **stack**'s **top**,
- **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 **stack**. 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.

```

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

```

Fig. 13.10 | Standard library **stack** adaptor class. (Part 1 of 2.)

20. “`std::stack`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/stack>.

```

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 << "\n\nPopping from vectorStack: ";
42     popElements(vectorStack);
43     std::cout << "\n\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 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.

The generic lambda `popElements` (lines 18–23) pops the elements off a 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.

13.12.2 queue Adaptor

A queue is similar to a waiting line. The first in line is the first 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.²¹ 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,
- **emplace** to construct an element in place at the queue’s **top**,
- **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 **push** to add elements to the queue. The **while** statement in lines 17–20 uses **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 **front** to read (but not remove) the queue’s first element for output. Line 19 removes the queue’s first element with **pop**, which is available in all adaptor classes.

```

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

```

Fig. 13.11 | Standard library queue adaptor class template. (Part 1 of 2.)

21. “queue.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/queue>.

```

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

Fig. 13.11 | Standard library queue adaptor class template. (Part 2 of 2.)

13.12.3 priority_queue Adaptor

Class **priority_queue** (from header `<queue>`) 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**.²² Elements added to a **priority_queue** are inserted in **priority order**, such that the highest-priority element will be the first element removed. This is usually accomplished by arranging the elements as a **heap**²³ that always maintains its highest-priority element at the front of the data structure. Element comparisons are performed with **comparator function object** `less<T>` by default.

The **priority_queue** operations include

- **push** to insert an element at the appropriate location based on **priority order**.
- **emplace** to construct an element in place and re-sort the **priority_queue** into **priority order**,
- **pop** to remove the **priority_queue**'s **highest-priority** element.
- **top** to get a reference to the **priority_queue**'s **top** element—implemented by calling the underlying container's **front** member function.
- **empty** to determine whether the **priority_queue** is empty—implemented by calling the underlying container's **empty** member function.
- **size** to get the **priority_queue**'s number of elements—implemented by calling the underlying container's **size** member function.

Figure 13.12 demonstrates **priority_queue**. Line 7 instantiates a **priority_queue** of **doubles** and uses a **vector** as the underlying data structure. Lines 10–12 use member function **push** to add elements to the **priority_queue**. Lines 17–20 use the **empty** member function (available in all containers) to determine whether the **priority_queue** is empty (line 17). If not, line 18 uses **priority_queue** function **top** to retrieve the **highest-priority** element (i.e., the largest value) in the **priority_queue** for output. Line 19 calls **pop**, available in all adaptor classes, to remove the **priority_queue**'s **highest-priority** element.

22. “`std::priority_queue`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/container/priority_queue.

23. Not to be confused with the heap for dynamically allocated memory. See Section 14.4.14.

```

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

Fig. 13.12 | Standard library priority_queue adaptor class.

13.13 bitset Near Container

Class **bitset** (header `<bitset>`) makes it easy to create and manipulate **bit** for representing a set of bit flags. **bitsets** are fixed in size at compile-time. Class **bitset** is an alternative tool for bit manipulation (see the appendix at <https://deitel.com/cpphtp11>).

The following declaration creates **bitset** **b**, in which every one of the **size** bits is initially 0 (“off”):

```
bitset<size> b;
```

The statement

```
b.set(bitNumber);
```

sets bit **bitNumber** “on.” You also can call this function with a **bool** second argument specifying whether to set the bit “on” (**true**) or “off” (**false**). The expression **b.set()** sets all bits in **b** “on.”

The statement

```
b.reset(bitNumber);
```

sets bit **bitNumber** “off.” The expression **b.reset()** sets all bits in **b** “off.”

The statement

```
b.flip(bitNumber);
```

toggles bit **bitNumber**—if the bit is “on,” **flip** sets it “off,” and vice versa. The expression **b.flip()** flips all bits in **b**.

The expression

```
b[bitNumber]
```

for a non-const `bitset` returns a `std::bitset::reference`²⁴ that enables you to manipulate the `bool` at position `bitNumber`; otherwise, it returns a copy of the `bool`.

The expression

```
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 or `false` if it's off. Otherwise, `test` throws an `out_of_range` exception.

The expression

```
b.size()
```

returns the number of bits.

The expression

```
b.count()
```

returns the number of bits that are set (`true`).

The expression

```
b.any()
```

returns `true` if any bit is set.

The expression

```
b.all()
```

returns `true` if all of the bits are set (`true`).

The expression

```
b.none()
```

returns `true` if none of the bits is set (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 `^=` can be used to combine `bitsets`. For example,

```
b &= b1;
```

performs a bit-by-bit AND between `b` and `b1`, setting each bit in `b` “on” if it's “on” in both `b` and `b1`. Bitwise OR

```
b |= b1;
```

sets each bit in `b` “on” if it's “on” in either or both of `b` and `b1`. Bitwise XOR

```
b ^= b2;
```

sets each bit in `b` “on” if it's “on” in only `b` and `b1`, but not both.

24. “`std::bitset<N>::reference`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/utility/bitset/reference>.

You also can perform a bitwise NOT operation. The following expression returns a copy of the `bitset` with all its bits flipped:

```
~b
```

The statement

```
b >>= n;
```

shifts the bits right by `n` positions. The expression

```
b <<= n;
```

shifts the bits left by `n` positions.

The expressions

```
b.to_string()  
b.to_ulong()  
b.to_ullong()
```

convert the `bitset` to a `string`, an `unsigned long` or an `unsigned long long`, respectively.

13.14 Wrap-Up

We began this chapter with two special computer science sections introducing Big O notation for expressing how hard algorithms have to work based on the number of items they process, and hash tables for high-speed storage and retrieval of data in applications with common real-world characteristics.

Next, we presented the three key components of the standard library—containers, iterators and algorithms. You learned about the linear sequence containers, `array` (Chapter 6), `vector` (Chapter 6 and this chapter), `deque`, `forward_list` and `list`, which all represent linear data structures. We discussed the nonlinear associative containers, `set`, `multiset`, `map` and `multimap` 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 near 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 function-call (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.

Self-Review Exercises

13.1 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) `deques` offer rapid insertions and deletions at front or back and direct access to any element.
- b) `lists` offer rapid insertion and deletion anywhere.
- c) `multimaps` offer one-to-many mapping with duplicates allowed and rapid key-based lookup.
- d) Associative containers are nonlinear data structures that typically can locate elements stored in the containers quickly.
- e) The container member function `cbegin` returns an `iterator` that refers to the container's first element.
- f) The `++` operation on an iterator moves it to the container's next element.
- g) The unary `*` operator when applied to a `const` iterator returns a `const` reference to the container element, allowing the use of non-`const` member functions.
- h) Function `capacity` returns the number of elements that can be stored in a `vector` before the `vector` needs to dynamically resize itself to accommodate more elements.
- i) 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` adaptor.
- j) `push_front` is available only for class `list`.
- k) Insertions and deletions can be made only at the front and back of a `map`.
- l) Class `queue` implements a first-in, first-out data structure that enables insertions at the front of the underlying container and deletions from the back.

13.2 Fill in the blanks in each of the following statements:

- a) The three key components of the “STL” portion of the Standard Library are _____, _____ and _____.
- b) Built-in arrays can be manipulated by Standard Library algorithms, using _____ as iterators.
- c) The Standard Library container adapter most closely associated with the last-in, first-out (LIFO) insertion-and-removal discipline is the _____.
- d) The sequence containers and _____ containers are collectively referred to as the first-class containers.
- e) The _____ container member function returns `true` if there are no elements in the container; otherwise, it returns `false`.
- f) The _____ container member function moves the elements of one container into another—this avoids the overhead of copying each element of the argument container.
- g) The container member function _____ is overloaded to return either an `iterator` or a `const_iterator` that refers to the first element of the container.
- h) Operations performed on a `const_iterator` return _____ to prevent modification to elements of the container being manipulated.
- i) The sequence containers are `array`, `vector`, `deque`, _____ and _____.
- j) Choose the _____ container for the best random-access performance in a container that can grow.

- k) Function `push_back`, which is available in sequence containers other than _____, adds an element to the end of the container.
- l) `vector` member functions `crbegin` and `crend` return _____ that represent the starting and ending points when iterating through a container in reverse.
- m) A unary _____ function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.
- n) The primary difference between the ordered and unordered associative containers is _____.
- o) The primary difference between a `multimap` and a `map` is _____.
- p) The `map` associative container performs fast storage and retrieval of unique keys and associated values. A single value can be associated with each unique key. This is called a(n) _____ mapping.
- q) Class _____ provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure.
- r) The C++20 `copy` algorithm from the _____ namespace receives only the range to copy and the output iterator representing where to copy the range's elements.
- s) The `std::ranges` algorithms typically are overloaded with a version that takes a range object and a version that takes a(n) _____ and a(n) _____.
- 13.3** Write a statement or expression that performs each of the following `bitset` tasks:
- Write a declaration that creates `bitset` `flags` of size `size`, in which every bit is initially 0.
 - Write a statement that sets bit `bitNumber` of `bitset` `flags` “off.”
 - Write a statement that returns a reference to the bit `bitNumber` of `bitset` `flags`.
 - Write an expression that returns the number of bits that are set in `bitset` `flags`.
 - Write an expression that returns `true` if all of the bits are set in `bitset` `flags`.
 - Write an expression that compares `bitsets` `flags` and `otherFlags` for inequality.
 - Write an expression that shifts the bits in `bitset` `flags` left by `n` positions.

Answers to Self-Review Exercises

13.1 a) True. b) False. They are doubly linked lists. c) True. d) True. e) False. It returns a `const_iterator`. f) True. g) False. Disallowing the use of non-`const` member functions. h) True. i) True. j) False. It's also available for class `deque`. k) False. Insertions and deletions can be made anywhere in a `map`. l) False. Insertions may occur only at the back and deletions may occur only at the front.

13.2 a) containers, iterators and algorithms. b) pointers. c) `stack`. d) associative. e) `empty`. f) move version of `operator=`. g) `begin`. h) `const` references. i) `list` and `forward_list`. j) `vector`. k) `array`. l) `const_reverse_iterators`. m) predicate. n) the unordered ones do not maintain their keys in sorted order. o) a `multimap` allows multiple values for each key and a `map` allows one value per key. p) one-to-one. q) `priority_queue`. r) `std::ranges`. s) iterator, sentinel.

13.3

- a) `bitset<size> flags;`
- b) `flags.reset(bitNumber);`
- c) `flags[bitNumber];`

- d) `flags.count()`
- e) `flags.all()`
- f) `flags != otherFlags`
- g) `flags <= n;`

Exercises

- 13.4** State whether each of the following is *true* or *false*. If *false*, explain why.
- a) Many of the Standard Library algorithms can be applied to various containers independently of the underlying container implementation.
 - b) arrays are fixed in size and offer direct access to any element.
 - c) `forward_lists` are singly linked lists, that offer rapid insertion and deletion only at the front and the back.
 - d) sets offer rapid lookup and duplicates are allowed.
 - e) In a `priority_queue`, the lowest-priority element is the first element removed.
 - f) The sequence containers represent non-linear data structures.
 - g) The `swap` non-member function swaps the contents of its two arguments (which must be of different container types) using move operations rather than copy operations.
 - h) Container member function `erase` removes all elements from the container.
 - i) An object of type `iterator` refers to a container element that can be modified.
 - j) We use `const` versions of the iterators for traversing read-only containers.
 - k) Insertions and deletions in the middle of a `deque` are optimized to minimize the number of elements copied, so it's more efficient than a `vector` but less efficient than a `list` for this kind of modification.
 - l) Container `set` does not allow duplicates.
 - m) The last-in, first-out data structure `stack` (from header `<stack>`) enables insertions into and deletions from the underlying data structure at one end.
 - n) Function `empty` is available in all containers except the `deque`.
- 13.5** Fill in the blanks in each of the following statements:
- a) The three styles of container classes are first-class containers, _____ and near containers.
 - b) Containers are divided into four major categories—sequence containers, ordered associative containers, _____ and container adapters.
 - c) The Standard Library container adapter most closely associated with the first-in, first-out (FIFO) insertion-and-removal discipline is the _____.
 - d) Built-in arrays, `bitsets` and `valarrays` are all _____ containers.
 - e) The _____ container member function returns the number of elements currently in the container.
 - f) The _____ container member function returns `true` if the contents of the first container are not equal to the contents of the second; otherwise, returns `false`.
 - g) We use iterators with sequences—these can be input sequences or output sequences, or they can be _____.
 - h) The Standard Library algorithms operate on container elements indirectly via _____.

- i) Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a(n) _____.
- j) Function _____ is available in *every* first-class container (except `forward_list`) and it returns the number of elements currently stored in the container.
- k) It can be wasteful to double a `vector`'s size when more space is needed. For example, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added, leaving 999,999 unused elements. You can use _____ and _____ to control space usage better.
- l) You can ask a `vector` or `deque` to return unneeded memory to the system by calling member function _____.
- m) The associative containers provide direct access to store and retrieve elements via keys (often called search keys). The ordered associative containers are `multiset`, `set`, _____ and _____.
- n) Classes _____ and _____ provide operations for manipulating sets of values where the values are the keys—there is *not* a separate value associated with each key.
- o) A `multimap` is implemented to efficiently locate all values paired with a given _____.
- p) The Standard Library container adapters are `stack`, `queue` and _____.

Discussion Questions

- 13.6** Why is it expensive to insert (or delete) an element in the middle of a `vector`?
- 13.7** Containers that support random-access iterators can be used with most but not all Standard Library algorithms. What is the exception?
- 13.8** Why would you use operator `*` to dereference an iterator?
- 13.9** Why is insertion at the back of a `vector` efficient?
- 13.10** When would you use a `deque` in preference to a `vector`?
- 13.11** What happens when you insert an element in a `vector` whose memory is exhausted?
- 13.12** When would you prefer a `list` to a `deque`?
- 13.13** What happens when the map subscript is a key that's not in the map?
- 13.14** Describe the `multiset` ordered associative container.
- 13.15** How might a `multimap` ordered associative container be used in a credit-card transaction processing system?
- 13.16** Write a statement that creates and initializes a `multimap` of strings and ints with three key-value pairs.
- 13.17** Explain the `push`, `pop` and `top` operations of a `stack`.
- 13.18** Explain the `push`, `pop`, `front` and `back` operations of a `queue`.
- 13.19** How does inserting an item in a `priority_queue` differ from inserting an item in virtually any other container?



Programming Exercises

13.20 (*Basic Descriptive Statistics*) In data science, you'll often use statistics to describe and summarize your data. Some basic descriptive statistics include:

- minimum—the smallest value in a collection of values.
- maximum—the largest value in a collection of values.
- range—the range of values from the minimum to the maximum.
- count—the number of values in a collection.
- sum—the total of the values in a collection.

Write a program that uses a sentinel-controlled inputs non-negative `int` values from the user and stores them in a `vector`. Calculate and display the minimum, maximum, range, count and sum of the `vector`'s elements.

13.21 (*Palindromes*) Write a function template `palindrome` that takes a `vector` parameter and returns `true` or `false` according to whether the `vector` does or does not read the same forward as backward (e.g., a `vector` containing 1, 2, 3, 2, 1 is a palindrome, but a `vector` containing 1, 2, 3, 4 is not).

13.22 (*Sieve of Eratosthenes with `bitset`*) This exercise revisits the *Sieve of Eratosthenes* for finding prime numbers that we discussed in Exercise 6.28. Use a `bitset` to implement the algorithm. Your program should display all the prime numbers from 2 to 1023, then allow the user to enter a number to determine whether that number is prime.

13.23 (*Sieve of Eratosthenes with `bitset` Modification*) Modify Exercise 13.22, the Sieve of Eratosthenes, so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number. Remember that a prime number's factors are only 1 and the prime number itself. Every nonprime number has a unique prime factorization. For example, the factors of 54 are 2, 3, 3 and 3. When these values are multiplied together, the result is 54.

13.24 (*map of Student Grades*) Write an application that represents an instructor's grade book as a `map` that maps each student's name (a `string`) to a `list` of `int` values containing that student's grades on three exams. Using the data stored in the `map`, calculate each student's average on the three exams and calculate the total class average. For example, given the student data:

- Susan: 92, 85, 100
- Eduardo: 83, 95, 79
- Azizi: 91, 89, 82
- Pantipa: 97, 91, 92

the program should produce the following output:

```
Average for Susan is 92.33
Average for Eduardo is 85.67
Average for Azizi is 87.33
Average for Pantipa is 93.33
Class's average is: 89.67
```

13.25 (Word Frequency Counting) Write a program that uses a map to count the number of occurrences of each word in a sentence entered by the user. Treat uppercase and lowercase letters the same and assume there is no punctuation in the sentence. The map's keys should be the unique words (strings), and its values will be integer counts of how many times each word appears in the sentence. You can break the sentence into its individual words (or tokens) using `std::string`'s `find` and `substr` functions that we introduced in Chapter 8. Given the following input

```
this is sample text with several words this is more sample text with
some different words
```

your program should produce the following output:

WORD	COUNT
different	1
is	2
more	1
sample	2
several	1
some	1
text	2
this	2
with	2
words	2
Number of unique words: 10	

13.26 (Character Counts) Use techniques similar to Exercise 13.25 to write a program that inputs a sentence from the user, then uses a map to summarize the number of occurrences of each letter. Ignore case, ignore blanks and assume the user does not enter any punctuation. Display a two-column table of the letters and their counts with the letters in sorted order.

13.27 (Duplicate Word Removal) Write a function that receives a list of words, then determines and displays in alphabetical order only the unique words. Treat uppercase and lowercase letters the same. The function should use a set to get the unique words in the list. Test your function with several sentences.

13.28 (Analyzing the Game of Craps) Modify the program of Fig. 5.5 to play 1,000,000 games of craps. Use a `wins` map to keep track of the number of games won for a particular number of rolls. Similarly, use a `losses` map to keep track of the number of games lost for a particular number of rolls. As the simulation proceeds, keep updating the maps.

A typical key-value pair in the `wins` dictionary might be 4 and 50217, indicating that 50217 games were won on the 4th roll. Display a summary of the results including:

- the percentage of the total games played that were won.
- the percentage of the total games played that were lost.
- the percentages of the total games played that were won or lost on a given roll (column 2 of the sample output).
- the cumulative percentage of the total games played that were won or lost up to and including a given number of rolls (column 3 of the sample output).

Your output should be similar to the following:

Percentage of wins: 50.2%		
Percentage of losses: 49.8%		
Percentage of wins/losses based on total number of rolls		
Rolls	% Resolved on this roll	Cumulative % of games resolved
1	30.10%	30.10%
2	20.80%	50.90%
3	14.10%	65.00%
4	9.90%	74.90%
5	7.40%	82.30%
6	4.60%	86.90%
7	3.70%	90.60%
8	2.40%	93.00%
9	1.90%	94.90%
10	1.10%	96.00%
11	0.90%	96.90%
12	0.80%	97.70%
13	0.80%	98.50%
14	0.30%	98.80%
15	0.30%	99.10%
16	0.30%	99.40%
17	0.50%	99.90%
25	0.10%	100.00%

Special Section: Systems Software Case Study—Building Your Own Compiler

In Chapter 7, we introduced the made-up Simpletron Machine Language (SML). In Exercise 7.10, you used simulation to create the Simpletron computer—a **virtual machine**—to execute programs written in SML. In this challenge section, you’ll build a “toy” compiler that converts programs written in **Simple**—a made-up, concise, high-level programming language—to SML. This section “ties” together key pieces of the programming process. You’ll:

- write several Simple high-level language programs,
- compile the programs using the compiler you’ll build, generating SML machine-language code into a file,
- load the SML machine-language code from that file into the Simpletron’s memory, and
- execute the SML machine-language programs on the Simpletron virtual machine you built in Exercise 7.10.

This section consists of six exercises. The first two cover some key computer-science technology you’ll need to implement your compiler. The third introduces the Simple high-level language with some completely coded examples and asks you to write several of your own Simple programs. The fourth guides you through building your **Simple compiler**. The fifth introduces the crucial topic of **compiler optimization**—you’ll modify your compiler to reduce the number of SML instructions it generates, which will make your SML programs more memory efficient and enable them to execute faster. The final exercise challenges you to modify your compiler to add more useful features.

13.29 (Infix-to-Postfix Converter) Compilers use stacks to help evaluate expressions and generate machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of single-digit integer constants, operators and parentheses. You can easily modify the algorithms we present to work with multiple-digit integers and floating-point numbers as well.

People generally write expressions like $3 + 4$ and $7 / 9$ with the operator between its operands. This is called **infix notation**. Computers “prefer” **postfix notation**, in which the **operator is written to the right of its two operands**. The preceding infix expressions would appear in postfix notation as $3\ 4\ +$ and $7\ 9\ /$.

To evaluate an infix expression, some compilers

- first convert the expression to postfix notation, then
- evaluate the postfix version.

Each of these **stack-oriented algorithms** requires one left-to-right pass of the expression. In this exercise, you’ll implement the **infix-to-postfix conversion algorithm**. In the next, you’ll implement the **postfix-expression evaluation algorithm**.

Write a program that converts a valid infix arithmetic expression with **single-digit integers** such as

$(6 + 2) * 5 - 8 / 4$

to a postfix expression. The postfix version of the preceding infix expression is

$6\ 2\ +\ 5\ *\ 8\ 4\ / -$

Note that postfix expressions contain no parentheses. The program should read the expression into **string infix** and create the postfix expression in **string postfix**.

The algorithm for creating a postfix expression is as follows:

1. Push a left parenthesis ‘(’ onto the stack.
2. Append a right parenthesis ‘)’ to the end of **infix**.
3. While the stack is not empty, read **infix** from left to right and do the following:
 - If **infix**’s current character is a digit, append it to **postfix**.
 - If **infix**’s current character is a left parenthesis, push it onto the stack.
 - If **infix**’s current character is an operator,
 - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to **postfix**.
 - Push the current character in **infix** onto the stack.
 - If **infix**’s current character is a right parenthesis,
 - Pop operators from the top of the stack and append them to **postfix** until a left parenthesis is at the top of the stack.
 - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

- + addition
- - subtraction
- * multiplication
- / division

In this exercise, it's helpful to display the stack's contents each time you modify it to confirm that you've implemented the algorithm correctly. Unlike a first-class container, the `std::stack` adaptor class does not have iterators, so you cannot display its contents. Recall that, by default, a `std::stack` stores its elements in a `std::deque`. As a first-class container, a `std::deque` has iterators, so you can display its contents. For this exercise, implement the stack as a `std::deque<char>`, using its `push_front` and `pop_front` member functions to push and pop elements.

Your solution should consist of `main` and the following functions:

- a) `convertToPostfix` receives a `string` representing the infix expression and returns a `string` representing its postfix equivalent.
- b) `isOperator` returns a `bool` indicating whether its `char` argument is an operator.
- c) `precedence` returns a `bool` indicating whether the precedence of its first `char` parameter `operator1` is greater than or equal to the precedence of its second `char` parameter `operator2`.
- d) `printStack` prints the `std::deque<char>`'s contents.

13.30 (Postfix-Expression Evaluator) Write a program that evaluates a valid postfix expression such as

6 2 + 5 * 8 4 / -

Read a `string` representing a postfix expression containing only single-digit operands, binary operators and no parentheses (recall that parentheses are eliminated during the infix-to-postfix conversion). The program should scan the postfix expression and evaluate it using the following algorithm:

1. While the end of the postfix expression `string` has not been encountered, read the expression from left to right:
 - If the current character is a digit,
 - Push its integer value onto the stack. The integer value of a digit character is its value in the computer's character set minus the value of the zero character ('0') in the computer's character set.
 - Otherwise, if the current character is an operator,
 - Pop the stack's two top elements into variables `x` and `y`.
 - Calculate `y` operator `x`.
 - Push the calculation's result onto the stack.
2. When end of the postfix expression `string` is encountered, pop the stack's top value. This is the postfix expression's result.

This algorithm supports only binary arithmetic operators. So in *Step 1*, if the operator is '/', the top of the stack is 2, and the next element in the stack is 8, you'd pop 2 into `x`, pop 8 into `y`, evaluate $8 / 2$, and push the result, 4, back onto the stack. This applies to each binary arithmetic operator. For this exercise, use a `std::deque<int>` as the stack.

The following arithmetic operations are allowed in an expression:

- + addition
- - subtraction
- * multiplication
- / division

Your solution should consist of `main` and the following functions:

- `evaluatePostfixExpression` receives a `string` representing the postfix expression, evaluates it and returns the result as an `int`.
- `calculate` receives two `int` operands and a `char` representing the operator, evaluates `operand1 operator operand2` and returns its `int` result.
- `printStack` prints the `std::deque<int>`'s contents.

13.31 (The Simple Programming Language—Writing Simple Programs) Before building the compiler, let's discuss a simple yet powerful, high-level language similar to early versions of the BASIC programming language. We call the language **Simple**. Every Simple statement consists of a line number and a Simple instruction. Line numbers must appear in ascending order. Each instruction begins with one of the following Simple commands: **rem**, **input**, **let**, **print**, **goto**, **if...goto** or **end**, as described in the following table. Simple evaluates only integer expressions using the `+`, `-`, `*` and `/` operators. These operators have the same precedence as in C++. Parentheses can change an expression's order of evaluation. Exercise 13.34 suggests enhancements to the Simple compiler. Several enhancements, such as adding floating-point capability, require modifications to the Simpletron virtual machine as well.

Command	Example statement	Description
rem	<code>50 rem this is a remark</code>	Text following the command rem is a comment for documentation purposes only and is ignored—no SML code is generated.
input	<code>30 input x</code>	Display a question mark to prompt the user to enter a single integer. Read that integer from the keyboard and store the integer in the variable <code>x</code> .
let	<code>80 let u = 4 * (j - 7)</code>	Assign to the variable <code>u</code> the value of <code>4 * (j - 7)</code> . An arbitrarily complex infix expression can appear to the right of the equal sign.
print	<code>10 print w</code>	Display the variable <code>w</code> 's value.
goto	<code>70 goto 45</code>	Transfer program control to the statement at line 45.
if...goto	<code>35 if i == z goto 80</code>	Compare the values of <code>i</code> and <code>z</code> for equality and transfer control to the statement at line 80 if the condition is true; otherwise, continue execution with the next statement.
end	<code>99 end</code>	Terminate program execution.

Additional Simple Language Rules

Simple also has the following language rules:

- The Simple compiler **recognizes only lowercase letters**—all characters in a Simple program should be lowercase.
- A **variable name is a single letter**. Multi-character variable names are not allowed, so Simple programs should document their variables in `rem` statements.
- Simple uses **only int variables**.

- Simple does not have variable declarations—merely mentioning a variable name in a program declares the variable and initializes it to zero.
- Simple's syntax does not allow string manipulation (reading a string, writing a string, comparing strings, etc.). Exercise 13.34 proposes this as an enhancement.
- Simple uses the conditional branch **if...goto statement** and the unconditional branch **goto statement** to alter a program's flow of control. If the condition in the if...goto statement is true, control transfers to the specified line number. The relational and equality operators `<`, `>`, `<=`, `>=`, `==` or `!=` are valid in an if...goto statement. Their precedence is the same as in C++.

Our compiler assumes that Simple programs are entered correctly. Exercise 13.34 asks you to enhance the compiler to perform **syntax-error checking**.

Simple Program Examples

Let's consider several Simple programs that demonstrate the language's features. The following Simple Program reads two integers from the keyboard, stores the values in variables `a` and `b`, and computes and prints their sum (stored in variable `c`). We've included empty `rem` statements for program readability.

```
10 rem    input two integers, then determine and print their sum
15 rem
20 rem    input the two integers
30 input a
40 input b
45 rem
50 rem    add the integers and store the result in c
60 let c = a + b
65 rem
70 rem    print the result
80 print c
90 rem    terminate program execution
99 end
```

```
? 5
? 3
8
```

The following program determines and prints the larger of two integers. The integers are input from the keyboard and stored in the variables `s` and `t`. The if...goto statement tests the condition `s >= t`. If the condition is true, control transfers to line 90, which displays `s`. Otherwise, the program displays `t`, then transfers control to the `end` statement in line 99, where the program terminates.

```
10 rem    input two integers, then determine and print the larger one
20 input s
30 input t
32 rem
35 rem    test if s is greater than or equal to t
40 if s >= t goto 90
45 rem
```

(continued...)

```

50 rem t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem s is greater than or equal to t, so print s
90 print s
99 end

```

```

? 3
? 7
7

```

```

? 7
? 3
7

```

Simple does not have repetition statements like C++'s `for`, `while` or `do...while`. However, you can simulate these using the `if...goto` and `goto` statements. The following Simple program uses a **sentinel-controlled loop** to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable `j`. If the value entered is the **sentinel -9999**, control transfers to line 99, where the program terminates. Otherwise, `k` is assigned the square of `j`, `k` is output to the screen and control is passed to line 20, where the next integer is input.

```

10 rem calculate squares of integers until user enters -9999 sentinel to end
20 input j
23 rem
25 rem test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem loop to get next j
60 goto 20
99 end

```

```

? 3
9

? 7
49

? -9999

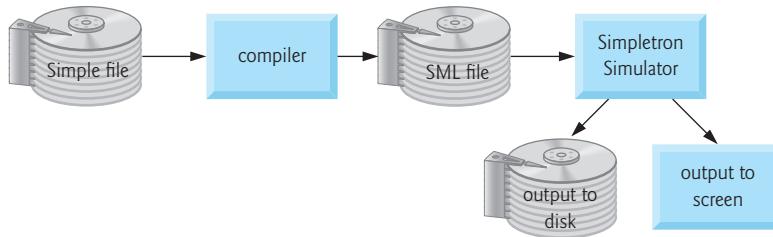
```

Write Your Own Simple Programs

Using the Simple programs shown in this exercise as your guide, write Simple programs to accomplish each of the following:

- a) Input three integers, determine their average and print the result.
- b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
- c) Use a counter-controlled loop to input seven integers, some positive and some negative, and compute and print their average.
- d) Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.
- e) Input 10 integers and print the smallest.
- f) Calculate and print the sum of the even integers from 2 to 30.
- g) Calculate and print the product of the odd integers from 1 to 9.

13.32 (Building a Compiler; Prerequisite: Complete Exercises 7.9, 7.10, 13.29, 13.30 and 13.31) Now that we've presented the Simple language (Exercise 13.31), let's discuss how to build a Simple compiler. The following diagram summarizes the process for compiling a Simple program into SML, then executing it in the Simpletron simulator:



The compiler reads a file containing a Simple program, **compiles** it into **SML code**, then writes the SML one instruction per line to a text file. Next, the Simpletron simulator **loads** the SML file into the Simpletron's 100-element memory array, executes the program and outputs the results to the screen and to a file. We also send all screen outputs to a file to make it easy to print a hard copy.

The Simpletron simulator you developed in Exercise 7.10 takes its input from the keyboard, not a file. You must modify the Simpletron to read from a file (using the file-processing techniques you learned in Chapter 8) so it can run the programs your Simple compiler produces.

The compiler performs **two passes** of a Simple program to convert it to SML:

- The **first pass** constructs a **symbol table** (discussed in detail below). The compiler stores in the symbol table every **line number**, **variable name** and **constant** of the Simple program. Each is stored with its **type** and its **location** in the final SML code. The **first pass** also produces the corresponding **SML instruction(s)** for each Simple statement. As you'll see, if the Simple program contains statements that transfer control to lines later in the program, the **first pass** results in an SML program containing some **incomplete instructions**.
- The **second pass** locates and completes the **unfinished instructions** and outputs the SML program to a file.

First Pass

The compiler begins by reading into memory the Simple program's first statement. The compiler separates the line into its individual **tokens** (i.e., "pieces" of a statement) for processing and compilation. You can determine a statement's tokens using `std::string`'s `find` and `substr` functions that we introduced in Chapter 8. Recall that every statement begins with a **line number** followed by a **command**. As the compiler breaks the rest of the statement into tokens, if a token is a **line number**, a **variable**, or a **constant**, it's placed in the **symbol table**. A **line number** is placed in the **symbol table** only if it's the **first token** in a statement—you'll soon see what the compiler does with line numbers that are targets of conditional or unconditional branches.

The `symbolTable` object is an `std::array` of `TableEntry` objects representing each **symbol** in the program. There's no restriction on the number of symbols that can appear in the program, so the `symbolTable` could be large—make it 200 elements for now. You can adjust its size once you have a working compiler.

`TableEntry` is declared as an aggregate `struct` type:

```
struct TableEntry {
    int symbol;
    char type; // 'C' (constant), 'L' (line number), 'V' (variable)
    int location; // 00 to 99
};
```

Each `TableEntry` contains three members:

- **symbol** is an integer containing a variable's character representation (again, **variable names are single characters**), a **line number**, or an integer **constant**. Recall that a character like '`x`' has an underlying integer value.
- **type** is a character indicating the symbol's type—'`C`' for a **constant**, '`L`' for a **line number**, or '`V`' for a **variable**.
- **location** contains the Simpletron **memory location** (**00** to **99**) associated with the symbol. Simpletron memory is an array of 100 integers in which **SML instructions** and **data** are stored. For a **line number**, the **location** is the Simpletron **memory array** element at which the Simple statement's SML instructions begin. For a **variable** or **constant**, the **location** is the Simpletron memory array element that stores the **variable** or **constant**. **Variables** and **constants** are allocated from location **99** of the Simpletron's memory downward. The first variable or constant is stored in location **99**, the next in location **98**, and so on.

Symbol Table

The **symbol table** plays an integral part in converting Simple programs to SML. We learned in Exercise 7.10 that an **SML instruction** is a signed **four-digit integer** that comprises two parts—the **operation code** and the **operand**. The operation code is determined by the Simple command. For example, the Simple command `input` corresponds to SML **operation code 10** (the SML `read` command), and the Simple command `print` corresponds to SML **operation code 11** (the SML `write` command). The **operand** is a **memory location** containing the **data** on which the **operation code** performs its task. For instance, the **operation code 10** reads a value from the keyboard and stores it in the **memory location** specified by the **operand**. The compiler searches `symbolTable` to determine the Sim-

plete memory location for each symbol so the corresponding location can be used to complete the SML instructions.

Compiling from Simple to SML

Each Simple statement's compilation process is based on the particular command. For example, after the line number in a **rem** statement is inserted in the symbol table, the compiler ignores the statement's remainder—a **rem** statement is for documentation purposes only—no SML code is generated. The **input**, **print**, **goto** and **end** statements correspond to the SML *read*, *write*, *branch* (to a specific location) and *halt* instructions. The compiler converts statements containing these Simple commands directly to SML. A **goto** statement may initially contain an **unresolved reference** if the specified line number refers to a statement later in the Simple program file. This is called a **forward reference**.

Compiling goto Statements

When a **goto** statement is compiled with an **unresolved reference**, the SML instruction must be flagged to indicate that the compiler's second pass must complete the instruction. The flags are stored in 100-element array of int values named **flags**, in which each element is initialized to -1. If the **memory location** to which a **line number** refers is not yet known (that is, it's not in the **symbol table**), its **line number** is stored in array **flags** in the element with the same subscript as the **incomplete instruction**. The **incomplete instruction's operand** is set temporarily to 00. For example, an **unconditional branch instruction** (making a **forward reference**) is left as +4000 until the compiler's second pass, which we'll describe shortly.

Compiling if...goto Statements

Compiling an **if...goto** or **let** statement is more complicated than other statements—each produces more than one SML instruction. For an **if...goto** statement, the compiler produces code to test the condition and possibly branch to another line. The result of the branch could be an **unresolved forward reference**. Each Simple relational and equality operator can be simulated using SML's *branch zero* and *branch negative* instructions (or possibly a combination of both).

Compiling let Statements

For a **let** statement, the compiler produces code to evaluate an **arbitrarily complex infix arithmetic expression** consisting of operators, single-letter integer variable names, integer constants and possibly parentheses. Expressions should separate each operand and operator with a space. Exercises 13.29 and 13.30 presented the **infix-to-postfix conversion algorithm** and the **postfix-evaluation algorithm** compilers use to evaluate expressions. Before building your compiler, you should complete those exercises. The compiler converts each expression from **infix notation** to **postfix notation**, then evaluates the **postfix expression**. As you'll see, the compiler actually generates machine-language instructions in the process of performing the postfix expression evaluation.

Compiling Expressions

How is it that the compiler produces the **machine language** to evaluate an expression containing **variables**? The **postfix-evaluation algorithm** contains a “**hook**” that allows our compiler to generate SML instructions rather than actually evaluating the expression.

To enable this “hook” in the compiler, the postfix-evaluation algorithm must be modified to:

- search the **symbol table** for each **symbol** it encounters (and possibly insert it),
- determine the symbol’s corresponding **memory location**, and
- **push the memory location instead of the symbol onto the stack.**

When an operator is encountered in the **postfix expression**, the stack’s top two memory locations are popped, and SML for implementing the operation is produced using the **memory locations as operands**. Each subexpression’s result is stored in a **temporary memory location** and **pushed back onto the stack**, so the postfix expression’s evaluation can continue. When **postfix evaluation is complete**, the **result’s memory location** is the **only location left on the stack**. This is popped, and SML instructions are generated to assign the result to the variable at the left of the `let` statement.

Second Pass

The compiler’s second pass performs two tasks:

- **resolve any unresolved references**, and
- **output the SML code to a file.**

Resolution of each reference occurs as follows:

1. Search the `flags` array for an **unresolved reference** (i.e., an element with a value other than `-1`).
2. Locate the element in `symbolTable` containing the symbol stored in the `flags` array (be sure that the type of the symbol is '`L`' for line number).
3. Insert the memory location from structure member `location` into the instruction with the **unresolved reference** (remember that an instruction containing an **unresolved reference has operand 00**).
4. Repeat *Steps 1–3* until the end of the `flags` array is reached.

After the resolution process is complete, the compiler outputs the SML code to a file with one SML instruction per line. The Simpletron can read this file and execute its instructions (after the simulator is modified to read its input from a file, of course).

A Complete Example

The following example illustrates a complete conversion of a Simple program to SML. Consider a Simple program that inputs an integer, sums the values from 1 to that integer and prints that sum. So, if the user enters 4, the program would calculate $1 + 2 + 3 + 4$, which is 10 and print that value. Figure 13.13 shows the program and the SML instructions produced by the compiler’s first pass. Figure 13.14 shows the symbol table constructed by the compiler’s first pass. Figure 13.15 shows how the compiler allocates Simpletron memory downward from cell 99. In a moment, we’ll do a step-by-step walk-through showing precisely how the compiler creates these tables.

Simple program	SML location and instruction	Description
5 rem sum 1 to x	<i>none</i>	rem ignored
10 input x	00 +1099	read x into location 99
15 rem check y == x	<i>none</i>	rem ignored
20 if y == x goto 60	01 +2098 02 +3199 03 +4200	load y (location 98) into accumulator sub x (location 99) from accumulator branch zero to unresolved location
25 rem increment y	<i>none</i>	rem ignored
30 let y = y + 1	04 +2098 05 +3097 06 +2196 07 +2096	load y (location 98) into accumulator add 1 (location 97) to accumulator store in temporary location 96 load from temporary location 96
	08 +2198	store accumulator in y (location 98)
35 rem add y to total t	<i>none</i>	rem ignored
40 let t = t + y	09 +2095 10 +3098 11 +2194 12 +2094	load t (location 95) into accumulator add y (location 98) to accumulator store in temporary location 94 load from temporary location 94
	13 +2195	store accumulator in t (location 95)
45 rem loop to y == x test	<i>none</i>	rem ignored
50 goto 20	14 +4001	branch to location 01
55 rem output result	<i>none</i>	rem ignored
60 print t	15 +1195	output t (location 95) to screen
99 end	16 +4300	terminate execution

Fig. 13.13 | SML instructions produced after the compiler's first pass.

Symbol	Type	Location
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97

Fig. 13.14 | Symbol table for program of Fig. 13.13. (Part I of 2.)

Symbol	Type	Location
<i>Temporary 96 allocated</i>		
35	L	09
40	L	09
't'	V	95
<i>Temporary 94 allocated</i>		
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 13.14 | Symbol table for program of Fig. 13.13. (Part 2 of 2.)

Data counter	Value	Type
...		
93		Next Simpletron memory cell to allocate
94	<i>none</i>	<i>Temporary variable</i>
95	't'	Variable
96	<i>none</i>	<i>Temporary variable</i>
97	1	Constant
98	'y'	Variable
99	'x'	Variable

Fig. 13.15 | Compiler allocates Simpletron memory downward from the last cell of memory (99).

As you look at the Simple program and corresponding SML instructions, recall that Simple does not have variable declarations, and the Simpletron initializes all memory locations to zero. So, when a variable is first mentioned, our compiler will create the variable and, if it's not initialized, it will have the value zero by default.

Most Simple statements convert directly to single SML instructions. The exceptions in this program are `rem` statements, the `if...goto` statement in line 20 and the `let` statements in lines 30 and 40. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a `goto` or an `if...goto` statement—each of these is allowed by Simple.

Line 20 of the program specifies that if the condition `y == x` is true, program control should transfer to line 60. Because line 60 appears *later* in the program, the compiler's `first pass` has not yet placed 60 in the symbol table—line numbers are placed in the symbol table only when they appear as the first token in a statement the compiler has processed. So, it's not yet possible to determine the operand of the SML branch-zero instruction at

location 03 in the array of SML instructions. The compiler places 60 in location 03 of the **flags array** to indicate that the second pass will complete this instruction.

We must keep track of the next instruction location in the SML array because **there is not a one-to-one correspondence between Simple statements and SML instructions**. For example, the **if...goto** statement of line 20 compiles into *three* SML instructions. Each time an instruction is produced, we must increment the instruction counter to the next SML array location. The Simpletron's limited memory size could present a problem for Simple programs with many statements, variables and constants. It's conceivable that the compiler could run out of Simpletron memory. To test for this case, your program should contain a **data counter** to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the data counter's value, the SML array is full. In this case, the compilation process should terminate, and the compiler should display an "out-of-memory" error message. Exercises 13.34 challenges you to modify the Simpletron's memory to hold 1000 cells numbered 000 to 999. This will help you compile more substantial Simple programs.

Step-by-Step Explanation of the Compilation Process's First Pass

Let's walk through the compilation process for the Simple program in Fig. 13.13.

Compiling Line 5

The compiler reads the first line of the program:

```
5 rem sum 1 to x
```

You can determine the tokens in the statement using `std::string`'s `find` and `substr` functions that we introduced in Chapter 8. Consider defining your own function to tokenize a string and return a `std::vector<std::string>` containing the tokens.

For each statement, the first token is the line number—in this case, "5"—which you can convert to an `int` with the `<string>` header's `stoi` function, so the symbol 5 can be placed in the symbol table. If the symbol is not found, it's inserted in the symbol table. Since we're at the beginning of the program and this is the first line, no symbols are in the table yet. So, 5 is inserted into the symbol table as type 'L' (line number) and assigned the first location in the SML memory array (00). Although this line is a remark, a space in the symbol table is allocated for the line number (in case it's the target of a `goto` or an `if...goto`). **If a program branches to a `rem` statement's line number, control resumes with the first executable statement after the `rem`.** No SML instruction is generated for a `rem` statement, so the instruction counter is not incremented.

Compiling Line 10

Next, the compiler tokenizes the statement

```
10 input x
```

The line number 10 is placed in the symbol table as type 'L' and assigned the first location in the SML array (00)—a remark (`rem`) began the program, so the instruction counter is still 00. The command `input` indicates that the next token is a variable (only a variable can appear as an argument in an `input` statement). Because `input` corresponds directly to an SML operation code, the compiler simply has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table. So, it's inserted into the

symbol table as the **ASCII representation of x**, given type 'V' (for variable), and assigned location 99 in the SML array. Data storage begins at 99 and is allocated downward—98, 97, and so on. SML code can now be generated for this statement. Operation code 10 (the SML **read** operation code) is multiplied by 100, and x's location (as determined in the symbol table) is added to it, which completes the instruction **+1099**. This is then stored in the SML array at location 00. The instruction counter is incremented by 1 because a single SML instruction was produced.

Compiling Line 15

Next, the compiler tokenizes the statement

```
15 rem check y == x
```

The symbol table is searched for line number 15, which is not found. The line number is inserted as type 'L' and assigned the SML array's next location (01). Again, **rem** statements do not produce code, so the instruction counter is not incremented.

Compiling Line 20

Next, the compiler tokenizes the statement

```
20 if y == x goto 60
```

Line number 20 is inserted in the symbol table and given type 'L' with the next location in the SML array, 01. The command **if** indicates that a condition is to be evaluated. The variable **y** is not found in the symbol table, so it's inserted and given the type 'V' and the SML location 98. Next, SML instructions are generated to evaluate the condition. There is no direct equivalent in SML for the **if...goto**, so it must be simulated by performing a calculation using **x** and **y** and branching based on the result. If **y** equals **x**, the result of subtracting **x** from **y** is zero. So, the SML **branch-zero instruction** can be used with the calculation result to simulate the **if...goto** statement.

The first step requires that **y** be loaded (from SML location 98) into the accumulator. This produces the instruction **01 +2098**. Next, **x** is subtracted from the accumulator. This produces the instruction **02 +3199**. The value in the accumulator may be zero, positive or negative. Since the **operator** is **==**, we want to **branch zero**. First, the symbol table is searched for the branch location (60), which is not found. So, 60 is placed in the **flags** array at location 03, and the instruction **03 +4200** is generated. We cannot add the branch location because we have not yet assigned a location to line 60 in the SML array—this location will be resolved later. The instruction counter is incremented to 04.

Compiling Line 25

The compiler proceeds to the statement

```
25 rem increment y
```

The line number 25 is inserted in the symbol table as type 'L' and assigned SML location 04. The instruction counter is not incremented.

Compiling Line 30

When the statement

```
30 let y = y + 1
```

is tokenized, the line number 30 is inserted in the symbol table as type 'L' and assigned SML location 04. Command `let` indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The constant integer 1 is inserted as type C and assigned SML location 97. Next, the right side of the assignment is **converted from infix to postfix notation**. Then the postfix expression ($y + 1$) is evaluated. Symbol y is located in the symbol table, and its corresponding memory location, 98, is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location, 97, is **pushed onto the stack**. When the operator $+$ is encountered, the postfix evaluator **pops the stack** into the right operand of the operator and **pops the stack** again into the left operand of the operator, then produces the SML instructions

```
04 +2098 (load y)
05 +3097 (add 1)
```

The result of incrementing y is stored in a **temporary location in memory** (96) with instruction

```
06 +2196 (store temporary)
```

and the temporary location is **pushed onto the stack**. Now that the expression has been evaluated, the result must be stored in the `let` statement's variable y . So, the temporary location is loaded into the accumulator, and the accumulator is stored in y with the instructions

```
07 +2096 (load temporary)
08 +2198 (store y)
```

Notice that some of these SML instructions—storing the accumulator into temporary location 96, then immediately reloading the accumulator from location 96—appear to be **redundant**. Eliminating such redundancy is an example of **compiler optimization**, which we'll say more about shortly.

Compiling Line 35

When the compiler tokenizes the statement

```
35 rem add y to total
```

it inserts line number 35 in the symbol table as type 'L' and assigns it location 09.

Compiling Line 40

The following statement is similar to line 30:

```
40 let t = t + y
```

The variable t is inserted in the symbol table as type 'V' and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions

```
09 +2095
10 +3098
11 +2194
12 +2094
13 +2195
```

are generated. The result of $t + y$ is assigned to temporary location 94 before being assigned to t (95). The instructions in memory locations 11 and 12 also appear to be redundant. Again, we'll discuss this **optimization** issue shortly.

Compiling Line 45

The statement

```
45 rem  loop to y == x test
```

is a remark, so line 45 is inserted in the symbol table as type 'L' and assigned SML location 14.

Compiling Line 50

The statement

```
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type 'L' and assigned SML location 14. The **equivalent of `goto` in SML** is the ***unconditional branch*** (40) **instruction** that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to produce the instruction +4001 at location 14.

Compiling Line 55

The statement

```
55 rem  output result
```

is a remark, so line 55 is inserted in the symbol table as type 'L' and assigned SML location 15.

Compiling Line 60

The statement

```
60 print t
```

is an output statement. Line number 60 is inserted in the symbol table as type 'L' and assigned SML location 15. The **SML equivalent of `print`** is **operation code 11 (write)**. Variable t 's location is determined from the symbol table, then added to the result of multiplying the operation code by 100. This forms the instruction +1195 at location 15.

Compiling Line 35

The statement

```
99 end
```

is the final line of the program. Line number 99 is stored in the symbol table as type 'L' and assigned SML location 16. The `end` command produces the SML instruction +4300 (43 is `halt` in SML). This is written as the final instruction in the SML memory array. The `halt` instruction has no operand.

The Compilation Process's Second Pass

On the compiler's *second pass*, we begin by searching the `flags` array for values other than -1. Location 03 contains 60, so the compiler knows that instruction 03 is incomplete. The

compiler completes the instruction by searching the symbol table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the compiler completes the instruction at location 03, replacing +4200 with +4215. The Simple program has now been compiled successfully.

Building Your Compiler

To build the compiler, you'll have to perform each of the following tasks:

- a) Modify the Simpletron simulator program you wrote in Exercise 7.10 to take its input from a file specified by the user (see Chapter 8). The simulator should also output its results to a file in the same format as the screen output.
- b) Modify the infix-to-postfix evaluation algorithm of Exercise 13.29 to process **multi-digit integer operands** and **single-letter variable-name operands**. Class `std::string`'s `find` and `substr` functions can be used to locate each constant and variable in an expression. Constants can be converted from `strings` to integers using the `<string>` header's `stoi` function. The postfix expression's data representation must be altered to support variable names and integer constants.
- c) Modify the postfix evaluation algorithm to process **multi-digit-integer operands** and **single-letter variable-name operands**. The algorithm also should now implement the previously discussed “hook” so that it produces SML instructions rather than directly evaluating the expression. Class `std::string`'s `find` and `substr` functions can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using the `<string>` header's `stoi` function. The data representation of the postfix expression must be altered to support variable names and integer constants.
- d) Build the compiler—incorporate *Part b* and *Part c* for evaluating expressions in `let` statements. Your program should contain a function that performs the compiler's **first pass** and one that performs its **second pass**.

13.33 (Optimizing the Simple Compiler) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, often in triplets called **productions**. A production normally consists of three instructions such as **load**, **add** and **store**. For example, here are five of the unoptimized SML instructions produced while compiling the program in Fig. 13.13:

```
04 +2098 (load)
05 +3097 (add)
06 +2196 (store)
07 +2096 (load)
08 +2198 (store)
```

The first three instructions are the production that adds 1 to *y*. Instructions 06 and 07 **store the accumulator value in temporary location 96**, then **load the value from that location right back into the accumulator** so instruction 08 can store the value in location 98. Often a production is followed by a load instruction for the same location that was just stored. This code can be *optimized* by eliminating the *store* instruction and the subsequent *load* instruction that operate on the same memory location. This optimization would decrease the SML program's “memory footprint” by 25% and improve its execu-

tion speed. Figure 13.16 shows the **optimized SML** for the program of Fig. 13.13. There are **four fewer instructions in the optimized code**. Modify your compiler to perform the optimization you learned in this exercise.

Simple program	SML location and instruction	Description
5 rem sum 1 to x	none	rem ignored
10 input x	00 +1099	read x into location 99
15 rem check y == x	none	rem ignored
20 if y == x goto 60	01 +2098 02 +3199 03 +4211	load y (98) into accumulator sub x (99) from accumulator branch to location 11 if zero
25 rem increment y	none	rem ignored
30 let y = y + 1	04 +2098 05 +3097 06 +2198	load y into accumulator add 1 (97) to accumulator store accumulator in y (98)
35 rem add y to total	none	rem ignored
40 let t = t + y	07 +2096 08 +3098 09 +2196	load t from location (96) add y (98) to accumulator store accumulator in t (96)
45 rem loop to y == x test	none	rem ignored
50 goto 20	10 +4001	branch to location 01
55 rem output result	none	rem ignored
60 print t	11 +1196	output t (96) to screen
99 end	12 +4300	terminate execution

Fig. 13.16 | Optimized code for the program of Fig. 13.13.

13.34 (Enhancing the Simple Compiler) Perform the following modifications to the Simple compiler. Some of these may also require modifications to the Simpletron Simulator program you wrote in Exercise 7.10. Many of these are quite challenging and could require substantial effort.

- Modify the Simpletron's memory to have **1000 cells (000–999)**. Modify the compiler to generate machine language appropriate for the 1000-element Simpletron memory.
- Allow the compiler to process **floating-point values** in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.
- Add support for **unary minus** to specify negative integer values.
- Allow the **modulus operator (%)** to be used in **let** statements. Modify the Simpletron Machine Language to include a modulus instruction.
- Allow **exponentiation** in a **let** statement using **^** as the exponentiation operator. Modify the Simpletron Machine Language to include an exponentiation instruction.

- f) Allow the compiler to recognize uppercase and lowercase letters in Simple statements. So, x and X would be treated as different variables. No modifications to the Simpletron Simulator are required.
- g) Allow **input** statements to read multiple variables' values, such as **input x, y**. No modifications to the Simpletron Simulator are required.
- h) Allow the compiler to **output multiple values in a single print statement**—such as **print a, b, c**—that separates each value from the next by one space. No modifications to the Simpletron Simulator are required.
- i) Allow the **print** statement's operand to be an **infix expression**.
- j) Add **syntax-checking** capabilities to the compiler so **error messages** are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron Simulator are required.
- k) Allow **integer arrays**. No modifications to the Simpletron Simulator are required.
- l) Allow **subroutines** specified by the Simple commands **gosub** and **return**. The **gosub** command passes program control to a subroutine, and the **return** command passes control back to the statement after the gosub. This is similar to a C++ function call. The same subroutine can be called from many gosubs distributed throughout a program. No modifications to the Simpletron Simulator are required.
- m) Allow **repetition structures** of the form

```
for x = 2 to 10
    rem Simple statements
    next
```

This **for** statement loops from 2 to 10 with a default **increment of 1**. No modifications to the Simpletron Simulator are required.

- n) Allow **repetition structures** of the form

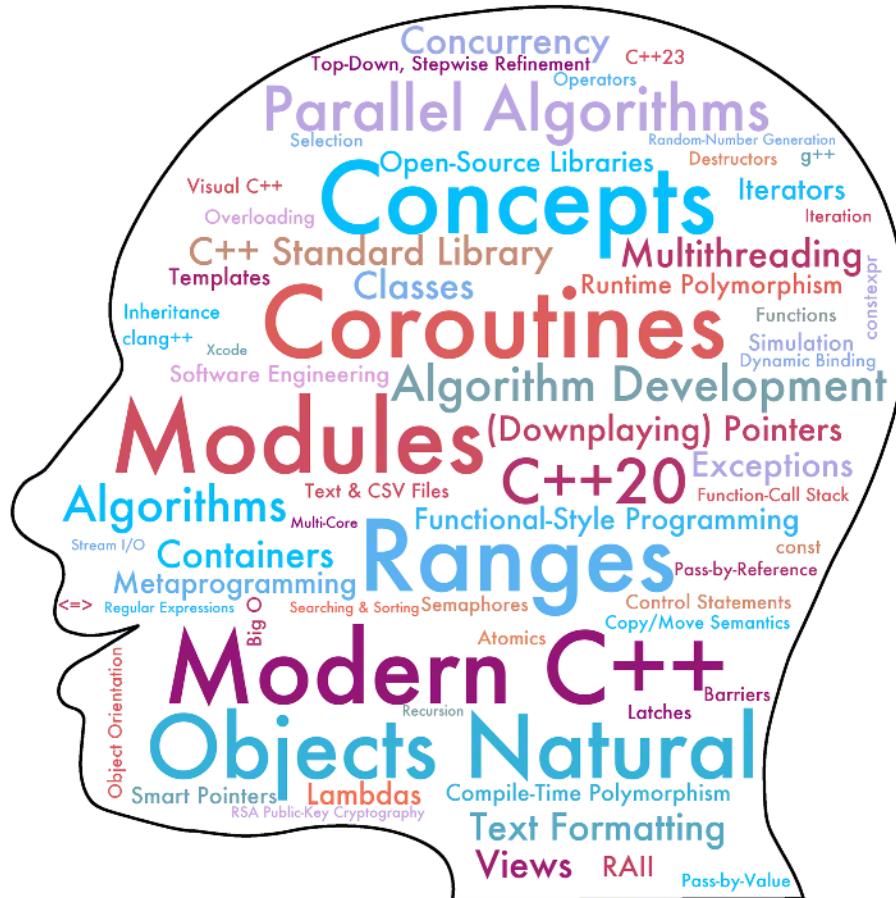
```
for x = 2 to 10 step 2
    rem Simple statements
    next
```

This **for** statement loops from 2 to 10 with an **increment of 2**. The **next** line marks the end of the **for** statement's body. No modifications to the Simpletron Simulator are required.

This page intentionally left blank

Standard Library Algorithms and C++20 Ranges & Views

14



Objectives

In this chapter, you'll:

- Understand minimum iterator requirements for working with standard library containers and algorithms.
 - Capture local variables for use in lambda expressions' bodies.
 - Use many of the C++20 `std::ranges` algorithms.
 - Understand C++20 Concepts corresponding to the C++20 `std::ranges` algorithms' minimum iterator requirements.
 - Compare `std::ranges` algorithms with their older common-range versions.
 - Use iterators with algorithms to access and manipulate container elements.
 - Pass lambdas, function pointers and function objects into standard library algorithms mostly interchangeably.
 - Use projections to transform objects in a range while processing them with C++20 range algorithms.
 - Use C++20 views and lazy evaluation with C++20 ranges.
 - Learn about possible C++23 ranges features.
 - Be introduced to parallel algorithms for performance enhancement—we'll discuss these in Chapter 17

Outline

14.1	Introduction	
14.2	Algorithm Requirements: C++20 Concepts	
14.3	Lambdas and Algorithms	
14.4	Algorithms	
14.4.1	<code>fill</code> , <code>fill_n</code> , <code>generate</code> and <code>generate_n</code>	
14.4.2	<code>equal</code> , <code>mismatch</code> and <code>lexicographical_compare</code>	
14.4.3	<code>remove</code> , <code>remove_if</code> , <code>remove_copy</code> and <code>remove_copy_if</code>	
14.4.4	<code>replace</code> , <code>replace_if</code> , <code>replace_copy</code> and <code>replace_copy_if</code>	
14.4.5	Shuffling, Counting, and Minimum and Maximum Element Algorithms	
14.4.6	Searching and Sorting Algorithms	
14.4.7	<code>swap</code> , <code>iter_swap</code> and <code>swap_ranges</code>	
14.4.8	<code>copy_backward</code> , <code>merge</code> , <code>unique</code> , <code>reverse</code> , <code>copy_if</code> and <code>copy_n</code>	
14.4.9	<code>inplace_merge</code> , <code>unique_copy</code> and <code>reverse_copy</code>	
14.4.10	Set Operations	
14.4.11	<code>lower_bound</code> , <code>upper_bound</code> and <code>equal_range</code>	
14.4.12	<code>min</code> , <code>max</code> and <code>minmax</code>	
14.4.13	Algorithms <code>gcd</code> , <code>lcm</code> , <code>iota</code> , <code>reduce</code> and <code>partial_sum</code> from Header <code><numeric></code>	
14.4.14	Heapsort and Priority Queues	
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	Future Ranges Enhancements	
14.11	Wrap-Up Exercises	

14.1 Introduction

This chapter is intended as a reference to the many standard library algorithms. It focuses 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 prepackaged, templated algorithms:

- 90 in the `<algorithm>` header's `std` namespace—82 also are overloaded in the **C++20 `std::ranges` namespace**,
- 11 in the `<numeric>` header's `std` namespace, and
- 14 in the `<memory>` header's `std` namespace—all 14 also are overloaded in the **C++20 `std::ranges` namespace**.

For the complete list of algorithms and their descriptions, visit

<https://en.cppreference.com/w/cpp/algorithm>

Minimum Algorithm Requirements and C++20 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 specify their requirements with **C++20 concepts**. 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

- the `std::ranges` namespace for use with C++20 ranges, and
- the `std` namespace for use with pre-C++20 common ranges—pairs of iterators representing a range’s first element and the element one past the range’s end.

We’ll focus mainly on the C++20 ranges versions.^{1,2}

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)`. Throughout this chapter, when we say that an algorithm can receive a function as an argument, we mean that it can receive a function, a lambda or a function object.

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 `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 parallel overloads for 69 standard library algorithms in the header `<algorithm>`, enabling you to enhance program performance on multi-core architectures. 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 17, Parallel Algorithms and Concurrency: A High-Level View, 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 in parallel on multiple cores so you can see the performance improvement.

Looking Ahead to the Future of Ranges

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 14.10 overviews updates expected in future C++ versions and mentions the open-source project `rangesnext`,³ which contains implementations for some proposed updates.

-
1. Tristan Brindle, “An Overview of Standard Ranges,” September 29, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=sYLgG7Q5Zws>.
 2. Tristan Brindle, “C++20 Ranges in Practice,” October 8, 2020. Accessed April 18, 2023. https://www.youtube.com/watch?v=d_E-VLyUnzc.
 3. Corentin Jabot, “Ranges for C++23.” Accessed April 18, 2023. <https://github.com/cor3ntin/rangesnext>.

14.2 Algorithm Requirements: C++20 Concepts



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 various containers—a strength of the standard library algorithms.

Container class templates and their corresponding iterator class templates typically reside in the same header. For example, the `<vector>` header contains the templates for `vector` 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



For maximum reuse, each algorithm can operate on any container that meets the algorithm's minimum iterator requirements.⁴ For example, an algorithm requiring **forward iterators** can operate on any container that provides *at least* forward iterators. All standard library algorithms can operate on **vectors** and **arrays** because they support contiguous iterators that provide every iterator operation discussed in Section 13.5.

Before C++20,

- each container's documentation mentioned the iterator level it supported, and
- each algorithm's documentation mentioned its minimum iterator requirements.

You were expected to adhere to an algorithm's documented requirements by using only containers with iterators that satisfied those requirements. If you passed the wrong iterator type, the compiler would substitute that type throughout the algorithm's template definition and, as Bjarne Stroustrup observed, produce "spectacularly bad error messages."⁵

C++20 Concepts

One of the "big four" C++20 features is **concepts**—a technology for constraining 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

When an algorithm's parameter is constrained by a concept, the compiler can check the requirements in the function 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.

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 in the

-
4. Alexander Stepanov and Meng Lee, "The Standard Template Library, Section 2: Structure of the Library," October 31, 1995. Accessed April 18, 2023. <http://stepanovpapers.com/STL/DOC.PDF>.
 5. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—1. A Bit of Background," January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.
 6. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—Conclusion," January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.
 7. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces," January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.

standard.^{8,9} 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' function template signatures to specify the algorithms' iterator and range requirements.

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 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

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** introduced in Chapter 13. For a complete list of **iterator concepts**, see the "C++20 iterator concepts" section at

<https://en.cppreference.com/w/cpp/iterator>

C++20 Range 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`

8. "Index of Library Concepts." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/conceptindex>.

9. The standard also specifies 30 "exposition-only" concepts used only for discussion purposes. "Exposition-Only in the C++ Standard?" Answered December 28, 2015. Accessed April 18, 2023. <https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard>.

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 many standard library algorithms' behavior by passing a function as an argument. You saw in Section 6.14.2 that **lambda expressions define anonymous functions**—usually, inside other functions.¹⁰ In Section 8.19, you saw that they can manipulate an enclosing function's local variables. We'll typically pass lambdas to standard library algorithms because they're convenient for expressing small tasks.

Figure 14.1 (which we discuss in pieces here) revisits the `copy` and `for_each` algorithms using the C++20's `std::ranges` versions. Recall that `for_each` receives as one of its arguments a function specifying a task to perform on each container element.

```

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
```

Fig. 14.1 | Lambda expressions.

Algorithm copy and Common Ranges Iterator Requirements vs. C++20 Ranges Iterator Requirements

Line 9 creates an array of `int` values, which line 13 displays using an `ostream_iterator` and the `copy` algorithm¹¹ from C++20's `std::ranges` namespace. Section 13.8.2 called the **common ranges** `std::copy` algorithm as follows:

```
std::copy(integers.cbegin(), integers.cend(), output);
```

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.

- 10. Lambdas also can be defined at class or namespace scope. For details, see <https://en.cppreference.com/w/cpp/language/lambda>.
- 11. “`std::ranges::copy`, `std::ranges::copy_if`, `std::ranges::copy_result`, `std::ranges::copy_if_result`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/algorithm/ranges/copy>.

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)
```

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`, and the second must be an `output iterator`. Here, we write to the standard output stream, but we also could write into another range. The `output iterator` has several requirements:

- It must be `std::weakly_incrementable`, meaning it must support the `++` operator to enable iteration through a sequence. All iterators support this operator. Concepts
- It also must be `std::indirectly_copyable`, which specifies a relationship between the iterator for `copy`'s `input_range` and the `output iterator`. In particular, the `input_range`'s iterator must be `std::indirectly_readable`, enabling `copy` to dereference the iterator to read an element of a given type, and the `output iterator` must be `std::indirectly_writable`, enabling `copy` to dereference the iterator to write the copied element of that type into the target range. Concepts Concepts Concepts Concepts

The concepts `indirectly_readable` and `indirectly_writable`, in turn, have many additional requirements. For simplicity going forward, we'll say `output iterator` when an algorithm requires a `weakly_incrementable` output iterator. Concepts

Prefer C++20's **range-based algorithms** to 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.  SE  Err

Algorithm for_each

This example uses the `for_each` algorithm from C++20's `std::ranges` namespace twice. The first call in line 17 multiplies each `values` element by 2 and displays the result.

```
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
```

The `std::ranges::for_each` algorithm's two arguments are

- an `input_range` (`values`) containing the elements to process and
- a function with one argument, which is allowed to modify its argument in the `input_range`.

 Concepts

The `for_each` algorithm repeatedly calls the function in its second argument, passing one element at a time from its `input_range` argument.

Lambda with an Empty Introducer

Line 17's lambda multiplies its parameter `i` by 2 and displays the result. Lambdas begin with the **lambda introducer** (`[]`), followed by a parameter list and a body. This **lambda introducer is empty**, so it does not capture any 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 we're iterating over `int` values, the compiler infers parameter `i`'s type as `int`.

The line 17 lambda, when specialized with `int`, is similar to the stand-alone function

```
void timesTwo(int i) {
    cout << i * 2 << " ";
```

Had we defined this function, we could have passed it to `for_each`, as in

```
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`] captures the local variable `sum` (defined in line 20) by reference (`&`), so the lambda can modify `sum`'s value.^{12,13,14} 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, and a compilation error would occur because a lambda's argument is treated as `const` by default.

```
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 value's elements is: 10

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.

-
12. C++ Core Guidelines, “F.50: Use a Lambda When a Function Won’t Do (to Capture Local Variables, or to Write a Local Function).” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-capture-vs-overload>.
 13. C++ Core Guidelines, “F.52: Prefer Capturing By Reference in Lambdas That Will Be Used Locally, Including Passed to Algorithms.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-reference-capture>.
 14. C++ Core Guidelines, “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 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-value-capture>.

It's better to specify which variables to capture. If you specify a list of variables to capture, each one preceded by & will be captured by reference. 

Lambda Return Types

The compiler can infer a lambda's return type from the return statement in its body. You also can specify a lambda's return type explicitly using **trailing return type** syntax:

```
[](parameterList) -> type {lambdaBody}
```

The trailing return type (-> type) is placed between the parameter list's closing right parenthesis and the lambda's body.

14.4 Algorithms

Sections 14.4.1–14.4.14 demonstrate many of the standard library algorithms. Most of the algorithms we present are the C++20 `std::ranges` versions.

14.4.1 `fill`, `fill_n`, `generate` and `generate_n`

Figure 14.2 demonstrates the C++20's `std::ranges` namespace's `fill`, `fill_n`, `generate` and `generate_n` algorithms:

- Algorithms `fill` and `fill_n` set every element or the first n elements in a range to a specific value.
- Algorithms `generate` and `generate_n` use a **generator function**¹⁵ to create values for every element or the first n elements in a range. The generator function takes no arguments and returns a value.

Lines 9–12 define a `nextLetter` function, which generates letters. We'll also implement this as a lambda so you can see the similarities.¹⁶ Line 15 defines `chars`—a 10-element `std::array` of `char` values that we'll manipulate in this example.

```

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 // returns the 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{};
```

Fig. 14.2 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`.

15. Not to be confused with a C++20 generator coroutine (Chapter 18, C++20 Coroutines).

16. Function `nextLetter` is not thread-safe. Chapter 17 discusses creating thread-safe functions.

fill Algorithm

Line 16 uses the C++20 `std::ranges::fill` algorithm to place '5' in every `chars` element. The first argument must be an `output_range` so that the algorithm can

- iterate through the `range` by incrementing its `iterator` with `++` and
- dereference the `iterator` to write a new value into the current element.

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. In this example's outputs, we use bold text to highlight each algorithm's changes to `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**

fill_n Algorithm

Line 23 uses the C++20 `std::ranges::fill_n` algorithm to place the character 'A' (the third argument) in `chars`' first five elements (specified by the second argument). The first argument must be at least an `output_iterator`. An array object supports `contiguous_iterators`, and `chars` is non-const, so calling `chars.begin()` returns an iterator that `fill_n` can use to write values into `chars`.

```
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's: **A** **A** **A** **A** **A** **5** **5** **5** **5**

generate Algorithm

Line 29 uses the C++20 `std::ranges::generate` algorithm to generate values for every `chars` element. The first argument must be an `output_range`. The second argument is a function that takes no arguments and returns a value. Function `nextLetter` (lines 9–12) defines a static local `char` variable `letter` and initializes it to 'A'. Line 11 returns the current `letter` value, postincrementing it for use in the next call to the function.

```
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

generate_n Algorithm

Line 35 uses the C++20 `std::ranges::generate_n` algorithm to place the result of each call to `nextLetter` in the five elements of `chars` starting from `chars.begin()`. The first argument must be at least an `input_or_output_iterator` that is `indirectly_writable`

 Concepts

```

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

Using the generate_n Algorithm with a Lambda

Lines 41–46 call the C++20 `std::ranges::generate_n` algorithm, passing a no-argument lambda (lines 42–45) that returns a generated letter. The compiler infers from the `return` statement that the lambda's return type is `char`.

```

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

14.4.2 equal, mismatch and lexicographical_compare

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.

```

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 <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
```

Fig. 14.3 | Algorithms `equal`, `mismatch` and `lexicographical_compare`.

equal Algorithm

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.

```

24 // compare a1 and a2 for equality
25 std::cout << std::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 << std::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 allow you to pass a function that customizes how elements should be compared. For example, you can pass to the **`equal`** algorithm a function that receives as arguments the two elements to compare and returns a `bool` value indicating whether they 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 objects 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.

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

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, `in1` is equal to the first range's sentinel, and `in2` is equal to the second range's sentinel. Line 33 infers the variable `location`'s type with `auto`. 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).

```
32 // check for mismatch between a1 and a3
33 auto location{std::ranges::mismatch(a1, a3)};
34 std::cout << std::format("a1 and a3 mismatch at index {} ({} vs. {})\n",
35     (location.in1 - a1.begin()), *location.in1, *location.in2);
36
```

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. When we initialize a variable with an algorithm's return value (as in line 33), we'll use **auto** to infer the type. To help you understand why, consider the `std::ranges::mismatch` algorithm's return type in line 33:

```
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

```
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. Class-template argument deduction (CTAD) also could be used to infer the type arguments. So, we could declare the variable's type simply as `std::ranges::mismatch_result`.

Lexicographical_compare Algorithm

The C++20 `std::ranges::lexicographical_compare` algorithm (lines 41–42) 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 range, 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 should be considered less than its second. Of course, `strings` are comparable with the relational and equality operators, so line 42's call to function `lexicographical_compare` could be replaced with `s1 < s2`. This algorithm is primarily intended for use with ranges that cannot be compared directly with one another, such as two `std::spans`.

```
37     std::string s1{"HELLO"};
38     std::string s2{"BYE BYE"};
39
40     // perform lexicographical comparison of c1 and c2
41     std::cout << std::format("\n{} < {}:\n", s1, s2,
42                           std::ranges::lexicographical_compare(s1, s2));
43 }
```

```
"HELLO" < "BYE BYE": false
```

14.4.3 remove, remove_if, remove_copy and remove_copy_if

Figure 14.4 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.

```

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

```

Fig. 14.4 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`.

remove Algorithm

Lines 12–14 initialize 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**, which supports **forward_iterators**. The iterators also must be **std::permutable**, enabling operations such as swapping and moving elements as this algorithm removes elements. A vector supports more powerful **random_access_iterators**, so it can work with any algorithm that requires lesser iterators. This algorithm does not destroy the eliminated elements. Instead, it places the remaining elements at the beginning of the container and returns a subrange specifying the elements that are no longer valid. That subrange should not be used, so its elements typically are erased from the container, as we'll discuss momentarily.

(C) Concepts
(C) Concepts

```

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 auto removed{std::ranges::remove(v1, 10)};
18 v1.erase(removed.begin(), removed.end());
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

```

Erase–Remove Idiom

Perf

Line 18 uses the `vector`'s `erase` member function to delete the subrange `removed`, representing the invalid `vector` elements. This reduces the `vector`'s size to its remaining number of elements. Line 20 outputs the updated contents of `v1`.

The `remove` and `erase` combination in lines 17 and 18 performs a technique known as the **erase–remove idiom**.¹⁷ You remove elements via `remove` or `remove_if`, then call the `vector`'s `erase` member function to eliminate the now unused elements.

The **common-range** `std::remove` and `std::remove_if` algorithms each return a single iterator pointing to the first invalid element in the `vector`. When using the **common-range algorithms**, you can perform the **erase-remove idiom** in one statement, as in

```
v1.erase(std::remove(v1.begin(), v1.end(), 10), v1.end());
```

Container member functions like `erase` do not yet support C++20 ranges.

C++20 `std::erase` and `std::erase_if`



To simplify the **erase-remove idiom**, C++20 added the `std::erase` and `std::erase_if` functions. Both are overloaded in the headers `<string>`, `<vector>`, `<deque>`, `<list>` and `<forward_list>`, and `std::erase_if` is overloaded in `<map>`, `<set>`, `<unordered_map>` and `<unordered_set>`. Each function performs the **erase-remove idiom** in a single function call that receives a given container as its first argument. For example, lines 17–18 can be replaced with

```
std::erase(v1, 10);
```

`remove_copy` Algorithm



Line 28 uses the `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.

```
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
```

Iterator Adapters: `back_inserter`, `front_inserter` and `inserter`

The second argument specifies an **output iterator** indicating where the copied elements will be written—typically another container. 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. Rather than preallocating memory in advance, you can use an **iterator adaptor** to **grow a container**, allowing it to allocate more elements as you insert new ones. In line 28, the expression `std::back_inserter(c1)` uses the `back_inserter` iterator adaptor (header `<iterator>`) to create a `back_insert_iterator`. When the algorithm uses this

17. “Erase–remove idiom.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Erase-remove_idiom.

iterator to insert an element in `c1`, the iterator calls the container's `push_back` function to place the element at the end of `c1`. If the container needs more space, it grows to accommodate the new element. A `back_insert_iterator` cannot be used with arrays because they are fixed-size containers and do not have a `push_back` function.



There are two other inserters:

- A `front_inserter` creates a `front_insert_iterator` that uses a container's `push_front` member function to insert an element at the beginning of a container.
- An `inserter` creates an `insert_iterator` that uses the container's `insert` member function to insert an element in the container specified by the adaptor's first argument at the location specified by the `iterator` in the adaptor's second argument.

`remove_if` Algorithm

Line 38 calls the C++20 `std::ranges::remove_if` algorithm to delete 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.



```

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 v3
37     auto greaterThan9{[](auto x) {return x > 9; }};
38     auto removed2{std::ranges::remove_if(v3, greaterThan9)};
39     v3.erase(removed2.begin(), removed2.end());
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

```

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 infers 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 a subrange of elements that are no longer valid. We use that subrange in line 39 to erase those elements in `v3`. Line 41 outputs the updated contents of `v3`. Lines 38–39 can be replaced with C++20's `std::erase_if`:

```
std::erase_if(v3, greaterThan9);
```

remove_copy_if Algorithm

Line 49 calls the C++20 `std::ranges::remove_copy_if` algorithm to copy the elements from its `input_range` argument `v4` for which a unary predicate function (the lambda at line 37) returns `true`. The second argument must be an `output iterator` so the element being copied can be written into the destination range. We use a `back_inserter` to insert the copied elements into the vector `c2`. Line 51 outputs the contents of `c2`.

```

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{};
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
```

14.4.4 replace, replace_if, replace_copy and replace_copy_if

Figure 14.5 demonstrates replacing values in a sequence using C++20's `std::ranges` algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.

```

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 }
```

Fig. 14.5 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.

replace Algorithm

Line 16 calls the C++20 `std::ranges::replace` algorithm to replace all 10s in the `input_range` `a1` with 100s. The first argument must support `indirectly_writable` iterators, so `replace` can dereference the iterators to replace values in the range. The array `a1` is non-const, so its `iterators` can be used to write into the array.

```

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
```

replace_copy Algorithm

Line 26 calls the C++20 `std::ranges::replace_copy` algorithm to copy all elements in the `input_range` `a2`, replacing each 10 with 100. The second argument must be an `output iterator` representing where to write the copied elements. The algorithm copies or replaces every element in the `input_range`, so we allocated `c1` with the same number of elements as `a2`, but we could have used an empty vector and a `back_inserter`.

 Concepts

```

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's 10s with 100s: 100 2 15 4 100 6
```

replace_if Algorithm

Line 36 calls the C++20 `std::ranges::replace_if` algorithm to replace each element in the `input_range` `a3` for which a `unary predicate function` (`greaterThan9` in line 35) returns `true`. The first argument must support `output iterators`, so `replace_if` can replace values in the range. Here, we replace each value greater than 9 with 100.

 Concepts

```

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     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
```

replace_copy_if Algorithm

Line 46 calls the C++20 `std::ranges::replace_copy_if` algorithm to copy all elements in the `input_range` `a4`, and if the `unary predicate function` returns true, replace their values. Here, 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`.

```
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's values greater than 9 with 100s: 100 2 100 4 100 6
```

14.4.5 Shuffling, Counting, and Minimum and Maximum Element Algorithms

Figure 14.6 demonstrates `shuffle`, `count`, `count_if`, `min_element`, `max_element` and `minmax_element` from C++20's `std::ranges` namespace. We also use the `transform` algorithm to cube values in a range.

```
1 // fig14_06.cpp
2 // Shuffling, counting, and minimum and maximum element algorithms.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <random>
8
9 int main() {
10     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     std::cout << "a1: ";
14     std::ranges::copy(a1, output);
```

Fig. 14.6 | Mathematical algorithms of the standard library. (Part I of 2.)

```
a1: 1 2 3 4 5 6 7 8 9 10
```

Fig. 14.6 | Mathematical algorithms of the standard library. (Part 2 of 2.)

shuffle Algorithm

Line 18 calls the C++20 `std::ranges::shuffle` algorithm to randomly reorder `a1`'s elements. This requires a `random_access_range` with `random_access_iterators` and thus can be used with the containers `array`, `vector` and `deque` (and with built-in arrays). The algorithm's declaration indicates that the range's iterators must be `permutable`, which permits operations such as swapping and moving elements. A non-`const` array's iterators allow such operations. The `shuffle` algorithm's second argument is a `random-number-generator engine` (Section 5.8). Line 20 displays the shuffled results.

(C) Concepts

```
16    // create random-number engine and use it to help shuffle a1
17    std::default_random_engine randomEngine{std::random_device{}()};
18    std::ranges::shuffle(a1, randomEngine); // randomly order elements
19    std::cout << "\na1 shuffled: ";
20    std::ranges::copy(a1, output);
21
```

```
a1 shuffled: 5 4 7 6 3 9 1 8 10 2
```

count Algorithm

Line 27 calls the C++20 `std::ranges::count` algorithm to count the elements with a specific value (in this case, 8) in `a2`. The first argument must be an `input_range` so `count` can read elements in the range.

(C) Concepts

```
22    std::array a2{100, 2, 8, 1, 50, 3, 8, 8, 9, 10};
23    std::cout << "\n\na2: ";
24    std::ranges::copy(a2, output);
25
26    // count number of elements in a2 with value 8
27    auto result1{std::ranges::count(a2, 8)};
28    std::cout << "\nCount of 8s in a2: " << result1;
29
```

```
a2: 100 2 8 1 50 3 8 8 9 10
Count of 8s in a2: 3
```

count_if Algorithm

Line 31 calls the C++20 `std::ranges::count_if` algorithm to count in its `input_range` argument `a2` elements for which a unary predicate function returns `true`. Once again, we use a lambda to define a unary predicate that returns `true` for a value greater than 9.

(C) Concepts

```
30    // count number of elements in a2 that are greater than 9
31    auto result2{std::ranges::count_if(a2, [](auto x){return x > 9;})};
32    std::cout << "\nCount of a2 elements greater than 9: " << result2;
33
```

Count of a2 elements greater than 9: 3

min_element Algorithm

Line 35 calls the C++20 `std::ranges::min_element` algorithm to locate the smallest element in its `forward_range` argument a2. Such a range supports `forward_iterators`, which allow `min_element` to read the range's elements. The algorithm returns an iterator located at the first occurrence of the range's smallest element or its 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 two elements and returns `true` if the first should be considered 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, as in line 35.

```
34 // Locate minimum element in a2
35 if (auto result{std::ranges::min_element(a2)}; result != a2.end()) {
36     std::cout << "\n\na2 minimum element: " << *result;
37 }
38
```

a2 minimum element: 1

max_element Algorithm

Line 40 calls the C++20 `std::ranges::max_element` algorithm to locate the largest element in its `forward_range` argument a2. Such a range supports `forward_iterators`, which allow `max_element` to read the range's elements. The algorithm returns an iterator located at the first occurrence of the range's largest element or its sentinel if the range is empty. Again, you can provide a custom `binary predicate function` specifying how to compare the elements.

```
39 // Locate maximum element in a2
40 if (auto result{std::ranges::max_element(a2)}; result != a2.end()) {
41     std::cout << "\n\na2 maximum element: " << *result;
42 }
43
```

a2 maximum element: 100

minmax_element Algorithm

Line 45 calls the C++20 `std::ranges::minmax_element` algorithm to locate the smallest and largest elements in its `forward_range` argument a2. The algorithm returns a `std::ranges::minmax_element_result` containing iterators named `min` and `max` 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**.
- If there are duplicate largest elements, the **second iterator is located at the last of the largest values**—note that this is different from how `max_element` works.

You can provide a custom `binary predicate function` specifying how to compare elements.

```

44 // Locate minimum and maximum elements in a2
45 auto [min, max]{std::ranges::minmax_element(a2)};
46 std::cout << "\n a2 minimum and maximum elements: "
47     << *min << " and " << *max;
48

```

a2 minimum and maximum elements: 1 and 100

Structured Bindings

Line 45 uses the `minmax_element_result` returned by `minmax_element` to initialize the variables `min` and `max` using a **structured binding declaration**.^{18,19} 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, `std::array` object, `std::pair` or `std::tuple` (each of which has a size that's known at compile-time). **Structured bindings** also can be used to unpack the accessible data members of a `struct` or `class` object. In each Chapter 13 example in which a function returned a `std::pair`, we also could have used structured bindings to unpack its members.

transform Algorithm

Lines 51–52 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 range specified by the algorithm's second argument, which must be an **output 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. There is no guarantee of the order in which this function will be called for the range's elements, so it should not modify the `input_range`'s elements. To modify the original range's elements, use the `std::foreach` algorithm.²⁰

 Concepts

 Concepts

```

49 // calculate cube of each element in a1; place results in cubes
50 std::array<int, a1.size()> cubes{};
51 std::ranges::transform(a1, cubes.begin(),
52     [] (auto x){return x * x * x;});
53 std::cout << "\n a1 values cubed: ";
54 std::ranges::copy(cubes, output);
55 std::cout << "\n";
56 }

```

a1 values cubed: 125 64 343 216 27 729 1 512 1000 8

A `transform` overload accepts two `input_ranges`, an **output iterator** and a function that takes two arguments and returns a result. This version of `transform` passes corre-

 Concepts

- 18. “Structured Binding Declaration.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/structured_binding.
- 19. Dominik Berner, “Quick and Easy Unpacking in C++ with Structured Bindings.” May 24, 2018. Accessed April 18, 2023. <https://dominikberner.ch/structured-bindings/>.
- 20. “Algorithms Library—mutating Sequence Operations—Transform.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/alg.transform>.

sponding elements from each `input_range` to its function argument, then outputs the function's result via the `output 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.

```

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
```

Fig. 14.7 | Standard library search and sort algorithms.

find Algorithm

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 positioned at the first element containing the value; otherwise, it returns the range's sentinel (demonstrated by lines 24–29). We use the returned iterator in line 17 to calculate the index position at which the value was found.

```

15    // locate first occurrence of 16 in values
16    if (auto loc1{std::ranges::find(values, 16)}; loc1 != values.cend()) {
17        std::cout << "\n\nFound 16 at index: " << (loc1 - values.cbegin());
18    }
19    else { // 16 not found
20        std::cout << "\n\n16 not found";
21    }
22
23    // locate first occurrence of 100 in values
24    if (auto loc2{std::ranges::find(values, 100)}; loc2 != values.cend()) {
25        std::cout << "\nFound 100 at index: " << (loc2 - values.cbegin());
26    }
27    else { // 100 not found
28        std::cout << "\n100 not found";
29    }
30

```

```
Found 16 at index: 4
100 not found
```

find_if Algorithm

The C++20 `std::ranges::find_if` algorithm (line 35) performs an $O(n)$ linear search to locate the first value in its `input_range` argument (`values`) for which a **unary predicate function** (`isGreaterThan10` from line 32) returns `true`. The algorithm returns an iterator positioned at the first element containing a value for which the **predicate function** returns `true`; otherwise, it returns the range's sentinel.

 Concepts

```
31 // create variable to store lambda for reuse later
32 auto isGreaterThan10{[] (auto x){return x > 10;}};
33
34 // Locate first occurrence of value greater than 10 in values
35 auto loc3{std::ranges::find_if(values, isGreaterThan10)};
36
37 if (loc3 != values.cend()) { // found value greater than 10
38     std::cout << "\n\nFirst value greater than 10: " << *loc3
39     << "\nfound at index: " << (loc3 - values.cbegin());
40 }
41 else { // value greater than 10 not found
42     std::cout << "\n\nNo values greater than 10 were found";
43 }
44 }
```

```
First value greater than 10: 17
found at index: 2
```

sort Algorithm

The C++20 `std::ranges::sort` algorithm (line 46) 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 containers `array`, `vector` and `deque` (and with built-in 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 binary predicate returns `true`, the two elements are already in sorted order; otherwise, the two elements need to be reordered in the sequence.

 Concepts

```
45 // sort elements of values
46 std::ranges::sort(values);
47 std::cout << "\n\nvalues after sort: ";
48 std::ranges::copy(values, output);
49 }
```

```
values after sort: 2 5 7 8 10 11 13 16 17 20
```

binary_search Algorithm

The C++20 `std::ranges::binary_search` algorithm (line 51) performs an $O(\log n)$ binary search to determine whether the value 13 is in the `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 59 demonstrates a call to `binary_search` for which the value is not found. This algorithm also can receive a **binary predicate function** that takes two arguments and returns 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`.

```

50     // use binary_search to check whether 13 exists in values
51     if (std::ranges::binary_search(values, 13)) {
52         std::cout << "\n\n13 was found in values";
53     }
54     else {
55         std::cout << "\n\n13 was not found in values";
56     }
57
58     // use binary_search to check whether 100 exists in values
59     if (std::ranges::binary_search(values, 100)) {
60         std::cout << "\n\n100 was found in values";
61     }
62     else {
63         std::cout << "\n\n100 was not found in values";
64     }
65

```

```

13 was found in values
100 was not found in values

```

all_of Algorithm

The C++20 `std::ranges::all_of` algorithm (line 67) 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.

```

66     // determine whether all of values' elements are greater than 10
67     if (std::ranges::all_of(values, isGreaterThan10)) {
68         std::cout << "\n\nAll values elements are greater than 10";
69     }
70     else {
71         std::cout << "\n\nSome values elements are not greater than 10";
72     }
73

```

```

Some values elements are not greater than 10

```

any_of Algorithm

The C++20 `std::ranges::any_of` algorithm (line 75) 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 element in its `input_range` argument (`values`). If so, `any_of` returns true; otherwise, it returns false.

 Concepts

```

74    // determine whether any of values' elements are greater than 10
75    if (std::ranges::any_of(values, isGreaterThan10)) {
76        std::cout << "\n\nSome values elements are greater than 10";
77    }
78    else {
79        std::cout << "\n\nNo values elements are greater than 10";
80    }
81

```

Some values elements are greater than 10

none_of Algorithm

The C++20 `std::ranges::none_of` algorithm (line 83) 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.

 Concepts

```

82    // determine whether none of values' elements are greater than 10
83    if (std::ranges::none_of(values, isGreaterThan10)) {
84        std::cout << "\n\nNo values elements are greater than 10";
85    }
86    else {
87        std::cout << "\n\nSome values elements are greater than 10";
88    }
89

```

Some values elements are greater than 10

find_if_not Algorithm

The C++20 `std::ranges::find_if_not` algorithm (line 91) 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 positioned at the first element containing a value for which the predicate function returns false; otherwise, it returns the range's sentinel.

 Concepts

```

90  // locate first occurrence of value that is not greater than 10
91  auto loc4{std::ranges::find_if(not(values, isGreaterThan10)};
92
93  if (loc4 != values.cend()) { // found a value less than or equal to 10
94      std::cout << "\n\nFirst value not greater than 10: " << *loc4
95      << "\nfound at index: " << (loc4 - values.cbegin());
96  }
97  else { // no values less than or equal to 10 were found
98      std::cout << "\n\nOnly values greater than 10 were found";
99  }
100
101 std::cout << "\n";
102 }
```

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.

```

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

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 receives references to the two values being exchanged. In this case, we pass references to the array’s first and second elements.

```

15    std::swap(values[0], values[1]); // swap elements at index 0 and 1
16
17    std::cout << "\nafter std::swap of values[0] and values[1]: ";
18    std::ranges::copy(values, output);
```

```
after std::swap of values[0] and values[1]: 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.

```
20 // use iterators to swap elements at locations 0 and 1
21 std::iter_swap(values.begin(), values.begin() + 1);
22 std::cout << "\nafter std::iter_swap of values[0] and values[1]: ";
23 std::ranges::copy(values, output);
24
```

```
after std::iter_swap of values[0] and values[1]: 1 2 3 4 5 6 7 8 9 10
```

swap_ranges Algorithm

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.

 Concepts

 Concepts

```
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
```

Lines 38–39 call the C++20 `std::ranges::swap_ranges` overload specifying the portions of two **input_ranges** to swap. Here, we swap the first five elements of `values` with the first five elements of `values2`, indicating the two ranges to swap as **iterator pairs**:

 Concepts

- `values.begin()` and `values.begin() + 5` specify the first five elements in `values`.
- `values2.begin()` and `values2.begin() + 5` specify `values2`'s first five elements.

Specifying iterator pairs also works with the **common ranges version** in the `std` namespace. In this example, the two `ranges` are in different containers, but the `ranges` can be in the same container, in which case the `ranges` must not overlap.

```

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

```

14.4.8 `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`

Figure 14.9 demonstrates algorithms `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n` from C++20's `std::ranges` namespace.

```

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>
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 }
```

Fig. 14.9 | Algorithms `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`. (Part 1 of 2.)

```
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9
```

Fig. 14.9 | Algorithms `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`. (Part 2 of 2.)

copy_backward Algorithm

The C++20 `std::ranges::copy_backward` algorithm (line 21) copies its first argument's `bidirectional_range` (`a1`) into the range specified by the `output iterator` in its second argument—in this case, the end of the target container (`results.end()`). The algorithm copies the elements in reverse order, placing each into the target container starting from the element before `results.end()` and working toward the beginning of the container. The algorithm returns a `std::ranges::copy_backward_result` containing two iterators:

- The first is positioned at `a1.end()`.
- The second is positioned at the last element copied into the target range—that is, the beginning of `results` because the copy is performed backward.

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, which is the first element in the range.

```
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
```



move and move_backward Algorithm

You can use `move semantics` with ranges. 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

The C++20 `std::ranges::merge` algorithm (line 27) combines two `input_ranges` that are each sorted in ascending order and writes the results to the target container specified by the third argument's `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 should be considered less than the second for ordering purposes.

```

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

unique Algorithm

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 a subrange from the element after the last unique value through 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.

```

32  // eliminate duplicate values from v
33  std::vector v(results2.begin(), results2.end());
34  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

reverse Algorithm

The C++20 `std::ranges::reverse` algorithm (line 41) reverses the elements in its argument—a `bidirectional_range` that supports `bidirectional_iterators`.

```

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

copy_if Algorithm

The C++20 `std::ranges::copy_if` algorithm (lines 47–48) receives as arguments an `input_range`, an `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 a **std::ranges::in_out_result** containing two iterators:

- the first is an **input iterator** positioned at the end of the **input_range**, and
- the second is an **output iterator** positioned after the last element copied into the output container.

```
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

copy_n Algorithm

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 **output iterator** in the third argument—in this case, we use a **back_inserter** to add elements to a **vector**.

 Concepts

```
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

14.4.9 `inplace_merge`, `unique_copy` and `reverse_copy`

Figure 14.10 demonstrates algorithms `inplace_merge`, `unique_copy` and `reverse_copy` from C++20's **std::ranges** namespace.

```
1 // fig14_10.cpp
2 // Algorithms inplace_merge, unique_copy and reverse_copy.
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7 #include <vector>
8
```

Fig. 14.10 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part I of 2.)

```

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

Fig. 14.10 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 2 of 2.)

inplace_merge Algorithm

Line 18 calls the C++20 `std::ranges::inplace_merge` algorithm to merge two sorted ranges 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 should be considered less than the second.

```

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

unique_copy Algorithm

Line 24 calls the C++20 `std::ranges::unique_copy` algorithm to copy the unique elements in its first argument's sorted `input_range`—duplicates are ignored. The 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** specifying how to compare elements for equality.

```

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

reverse_copy Algorithm

Line 30 calls the C++20 `std::ranges::reverse_copy` algorithm to make a reversed copy of its first argument's `bidirectional_range`. The `output iterator` supplied as the second argument specifies where to place the copied elements—in this case, the `back_inserter` adds new elements to the vector `results2`, growing it as necessary.

```

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

14.4.10 Set Operations

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 we show, you can customize each algorithm's element comparisons by passing a `binary predicate function` that receives two elements and returns `true` if the first should be considered less than the second.

```

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 <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 }
```

Fig. 14.11 | Algorithms `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

includes Algorithm

Lines 26 and 30 each call the C++20 `std::ranges::includes algorithm` to compare two sorted `input_ranges` and determine whether every element of the second is in the first. Concepts (C) 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 range; otherwise, it returns `false`. In line 26, `a2`'s elements are all in `a1`, so `includes` returns `true`. In line 30, `a3`'s elements are not all in `a1`, so `includes` returns `false`.

```
24 // determine whether a2 is completely contained in a1
25 std::cout << std::format("\n\na1 {}\n a2",
26     std::ranges::includes(a1, a2) ? "includes" : "does not include");
27
28 // determine whether a3 is completely contained in a1
29 std::cout << std::format("\n\na1 {}\n a3",
30     std::ranges::includes(a1, a3) ? "includes" : "does not include");
31
```

```
a1 includes a2
a1 does not include a3
```

set_difference Algorithm

Line 34 calls the C++20 `std::ranges::set_difference algorithm` to find the elements from the first sorted `input_range` that are not in the second sorted `input_range`. Concepts (C) The ranges must be sorted using the same `comparison function`. The elements that differ are copied to the location specified by the third argument's `output iterator`—in this case, a `back_inserter` adds them to the vector `difference`.

```
32 // determine elements of a1 not in a2
33 std::vector<int> difference{};
34 std::ranges::set_difference(a1, a2, std::back_inserter(difference));
35 std::cout << "\n\nset_difference of a1 and a2 is: ";
36 std::ranges::copy(difference, output);
37
```

```
set_difference of a1 and a2 is: 1 2 3 9 10
```

set_intersection Algorithm

Lines 40–41 call the C++20 `std::ranges::set_intersection algorithm` to determine Concepts (C) 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 third argument's **output iterator**—in this case, a **back_inserter** adds them to the vector `intersection`.

```

38 // determine elements in both a1 and a2
39 std::vector<int> intersection{};
40 std::ranges::set_intersection(a1, a2,
41     std::back_inserter(intersection));
42 std::cout << "\n\nset_intersection of a1 and a2 is: ";
43 std::ranges::copy(intersection, output);
44

```

```
set_intersection of a1 and a2 is: 4 5 6 7 8
```

set_symmetric_difference Algorithm

Lines 48–49 call the C++20 `std::ranges::set_symmetric_difference` algorithm to determine the elements in the first sorted `input_range` that are not in the second sorted `input_range` and the elements in the second 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 third argument's **output iterator**—in this case, a **back_inserter** adds them to `symmetricDifference`.

C Concepts

```

45 // determine elements of a1 that are not in a3 and
46 // elements of a3 that are not in a1
47 std::vector<int> symmetricDifference{};
48 std::ranges::set_symmetric_difference(a1, a3,
49     std::back_inserter(symmetricDifference));
50 std::cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
51 std::ranges::copy(symmetricDifference, output);
52

```

```
set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15
```

set_union Algorithm

Line 55 calls the C++20 `std::ranges::set_union` algorithm to create a set of the elements 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 third argument's **output iterator**—in this case, a **back_inserter** adds them to `unionSet`. Elements that appear in both sets are copied only from the first set.

C Concepts

```

53 // determine elements that are in either or both sets
54 std::vector<int> unionSet{};
55 std::ranges::set_union(a1, a3, std::back_inserter(unionSet));
56 std::cout << "\n\nset_union of a1 and a3 is: ";
57 std::ranges::copy(unionSet, output);
58 std::cout << "\n";
59 }

```

```
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

14.4.11 lower_bound, upper_bound and equal_range

Figure 14.12 demonstrates the algorithms `lower_bound`, `upper_bound` and `equal_range` from C++20's `std::ranges` namespace. In addition to the capabilities we show, you can customize each algorithm's element comparisons by passing a **binary predicate function** that receives two elements and returns `true` if the first should be considered less than the second.

```

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 }
```

```
values 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.

Tower_bound Algorithm

Line 17 calls the C++20 `std::ranges::lower_bound` algorithm to find 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.

```

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 index: 5
```

upper_bound Algorithm

Line 22 calls the C++20 `std::ranges::upper_bound` algorithm to find in a sorted `forward_range` the last location at which the second argument could be inserted so the range would still be sorted. The algorithm returns an iterator pointing to that location.



```

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 index: 9

equal_range Algorithm

The C++20 `std::ranges::equal_range` algorithm (line 27) performs the `lower_bound` and `upper_bound` operations, then returns their results as a `std::ranges::subrange`, which we unpack into variables `first` and `last`.

```

26 // use equal_range to determine the lower and upper bound of 6
27 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 index: 5
 Upper bound of 6 is index: 9

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 where 3 can be inserted in order in `values`. Line 43 uses `upper_bound` to locate the last point where 7 can be inserted in order in `values`.

```

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 "
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()) << "\n";
46 }

```

Use `lower_bound` to locate the first point at which 3 can be inserted in order
 Lower bound of 3 is index: 2

Use `upper_bound` to locate the last point at which 7 can be inserted in order
 Upper bound of 7 is index: 9

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 (lines 8–12) each receive two arguments and return the minimum or maximum value. Each algorithm also has an overload that takes as a third argument a **binary predicate function** for **custom comparisons** determining whether the first argument should be considered less than the second.

```

1 // fig14_13.cpp
2 // Algorithms min, max and minmax.
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 'G' and 'Z' is: 'G'
 Maximum of 'G' and 'Z' is: 'Z'
 Minimum of 'z' and 'Z' is: 'Z'

Fig. 14.13 | Algorithms `min`, `max` and `minmax`.

`minmax` Algorithm with Two Arguments

The two-argument `minmax` algorithm (line 15) returns a pair of values containing the smaller and larger items, 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 a **custom comparison** determining whether the first argument should be considered less than the second.

```

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

```

```

Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12

```

minmax Algorithm for C++20 Ranges

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. Again, 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 should be considered less than the second.

```

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    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

```



14.4.13 Algorithms gcd, lcm, iota, reduce and partial_sum from Header <numeric>

Section 8.19.2 introduced the `accumulate` algorithm (header `<numeric>`). Figure 14.14 demonstrates `<numeric>` algorithms `gcd`, `lcm`, `iota`, `reduce` and `partial_sum`. This header's algorithms require **common ranges** and are expected to have ranges overloads in C++23.²¹

gcd Algorithm

The `gcd` algorithm (lines 14 and 15) receives two integer arguments and returns their greatest common divisor.

21. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2214r0.html#algorithms>.

```

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

```

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**.

```

17     // calculate the least common multiple of two integers
18     std::cout << "\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** representing a **common range** to fill. The last argument's type must support the `++` operator.

```

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) reduces 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 provides a custom initializer value (1) and a **binary function** that specifies how to perform the **reduction**. In this case, we used `std::multiplies{}` (header `<functional>`)—a predefined **binary function object**²² 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 the `operator()` function on the function object to produce a result. Any commutative and associative binary function that takes two values of the same type and returns a result of that type can be passed as the fourth argument. In Section 14.5, you'll see that header `<functional>` defines **binary function objects** for addition, subtraction, multiplication, division and modulus, among other operations.

```

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

```

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.²³

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 $1 + 2 + 3 + 4$)
- 15 (the sum $1 + 2 + 3 + 4 + 5$)

22. We say more about the predefined function objects in Section 14.5.

23. Sy Brand, “`std::accumulate` vs. `std::reduce`,” May 15, 2018. Accessed April 18, 2023. <https://blog.tartanllama.xyz/accumulate-vs-reduce/>.

```

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

```

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 $1 * 2 * 3 * 4 * 5$)

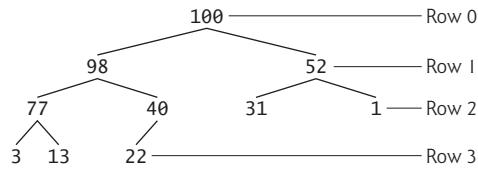
This `partial_sum` call is actually calculating the factorials of 1–5 (that is $1!$, $2!$, $3!$, $4!$ and $5!$).

14.4.14 Heapsort and Priority Queues

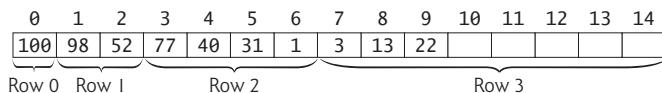
Section 13.12.3 introduced the `priority_queue` container adaptor. Elements added to a `priority_queue` are stored in a manner that enables removing them in **priority order**. The highest-priority element is always removed first—usually, the highest-priority element has the largest value, but this is customizable. Finding the highest-priority element can be accomplished 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 use of priority queues is in operating system process scheduling.

Heap Data Structure

A **heap** is commonly implemented as a **binary tree**. 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:



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 element at index 0.

Heap-Related Algorithms

Figure 14.15 demonstrates four `C++20 std::ranges` namespace algorithms related to heaps. First, we show `make_heap` and `sort_heap`, which implement the two steps in the **heapsort algorithm**, which has a worst-case runtime of $O(n \log n)$.²⁴

- *Step 1* arranges the elements of a container into a **heap**.
 - *Step 2* removes the elements from the **heap** to produce a sorted sequence.

Then, we show **push_heap** and **pop_heap**, which the **priority_queue** container adaptor²⁵ uses “under the hood” to maintain its **heap** as elements are added and removed.

```
1 // fig14_15.cpp
2 // Algorithms make_heap, sort_heap, push_heap and pop_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 make_heap, sort_heap, push_heap and pop_heap.

24. "Heapsort." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. <https://en.wikipedia.org/wiki/Heapsort>.

25. 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 April 18, 2023.

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.

```
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
```

make_heap Algorithm

Line 16 calls the C++20 `std::ranges::make_heap` algorithm to arrange 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`.

```
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
```

sort_heap Algorithm

Line 20 calls the C++20 `std::ranges::sort_heap` algorithm to sort the elements of its `random_access_range` argument. The range must already be a `heap`. Line 22 displays the sorted `heapArray`.

```
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
```

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 $O(\log n)$.²⁶ 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:

26. “Binary Heap.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Binary_heap.

- Line 28 appends one `int` to the vector argument `heap`.
- 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 last element is being added to the `heap` and that the 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` rearranges the elements into a heap.
- Line 31 displays the updated **heap data structure** after each value is added.

```

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

```

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:

- Line 38 calls the C++20 `std::ranges::pop_heap algorithm` to remove the `heap`'s largest value. The algorithm assumes that its `random_access_range` represents 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.

```

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

```

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, 52 and 100 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.

```
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
```

Next, line 53 calls the **pop lambda** to remove 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.

```
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.

```
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
```

Finally, lines 59–61 remove the heap's three remaining items. Note that after line 59 executes, the largest remaining element (22) becomes the `vector`'s first element.

```
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:
```

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 standard library algorithm that can receive a lambda or function pointer can also receive a **function object** (also called a **functor**)—that is, an object of a class that overloads the function-call operator (parentheses) with a function named **operator()**. The overloaded **operator()** must meet the algorithm's requirements for the number of parameters and the return type.



Function objects can be used syntactically and semantically like a lambda or a function pointer. The **operator()** function is invoked 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, so it can contain non-static data to maintain state for a specific **function object** or static data 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.²⁷
- Perhaps most importantly, the compiler can **inline function objects** for performance. A **function object's operator()** function is typically 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 implement lambdas as **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.²⁸



27. "Function Objects in the C++ Standard Library," March 15, 2019. Accessed April 18, 2023. <https://docs.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl>.

28. Scott Meyers. *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 Library

Header `<functional>` contains many predefined function objects. Each is implemented as a class template. The following table lists some of the commonly used standard library function objects. Each relational and equality function object has a corresponding one of the same name in the C++20 `std::ranges` namespace. Most are binary function objects that receive two arguments and return a result. Both `logical_not` and `negate` are unary function objects that receive one argument and return a result.

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		

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 into ascending order. We used the function object `less<T>` in Section 13.11's presentations to specify the order for keys in the ordered set and map containers.

You can see the complete list of function objects at

<https://en.cppreference.com/w/cpp/utility/functional>

and in the “Function Objects” section of the C++ standard.²⁹ The `std::ranges` algorithms that compare elements for ordering use `less<T>` as their default predicate function argument. 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—precisely the purpose of the `less<T>` function object.

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++23.³⁰ The `accumulate` algorithm has two overloads.

29. “General Utilities Library—function Objects.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/function.objects>.

30. Christopher Di Bella, “A Concept Design for the Numeric Algorithms,” August 2, 2019. Accessed April 18, 2023. <http://wg21.link/p1813r0>.

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:

- a **function pointer** to a **binary function** that takes two parameters of the **common range**'s element 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 element type and returns a result of that type, or
- a **lambda** that takes two parameters of the **common range**'s element type and returns a result of that type.

This example calls **accumulate** with a **function pointer**, then a **function object**, then a **lambda**.

```

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**.

```

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

```

Class SumSquaresClass

Lines 19–25 define class **SumSquaresClass**.³¹ Its overloaded **operator()** has two **int** parameters and returns an **int**. This meets the requirements for the **binary function** that **accumulate** can call when processing a **common range** of **ints**. Function **operator()** returns the sum of its first argument **total** and the square of its second argument **value**.

31. 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.

```

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

```

Calling Algorithm `accumulate`

We call `accumulate` three times:

- Lines 36–37 call `accumulate` with a pointer to `sumSquares` as its last argument.
- Lines 44–45 call `accumulate` with a `SumSquaresClass` function object as the last argument. `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`.

```

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

```

Each call to **accumulate** 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 prior 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 prior 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 prior 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

When working on objects containing 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` to output them conveniently.

```

1 // fig14_17.cpp
2 // Demonstrating projections with C++20 range algorithms.
3 #include <array>
4 #include <algorithm>
5 #include <format>
6 #include <iostream>
7 #include <iterator>
8 #include <string>
9 #include <string_view>
10

```

Fig. 14.17 | Demonstrating projections with C++20 range algorithms.

```

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 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, const Employee& e) {
26     out << std::format("{:10}{:10}{:10}", 
27         e.getLast(), e.getFirst(), e.getSalary());
28     return out;
29 }
30

```

Fig. 14.17 | Demonstrating projections with C++20 range algorithms.

Defining and Displaying an array<Employee>

Lines 32–36 define an array<Employee>, initializing it with three Employees. Line 41 displays the Employees, so we can confirm later that they’re sorted properly.

```

31  int main() {
32      std::array employees{
33          Employee{"Joo-won", "Wu", 5000},
34          Employee{"Amaia", "Jones", 7600},
35          Employee{"Titania", "Makaya", 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:
Wu      Joo-won    5000
Jones   Amaia     7600
Makaya Titania   3587

```

Using a Projection to Sort the array<Employee> in Ascending Order

Lines 45–46 call 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 `sort` 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—that is, `std::ranges::less`. 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.

```

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:
Makaya    Titania    3587
Wu         Joo-won    5000
Jones      Amaia      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

```
std::ranges::sort(employees, {}, &Employee::getSalary);
```

To be used as a projection, the member function must be **public** and must not be overloaded. Also, it must have no parameters because the `std::ranges` algorithms cannot receive additional arguments to pass to the member function specified in the projection.³²

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
```

³². “Under the hood,” `std::ranges::sort` uses the `std::invoke` function (header `<functional>`) to call the provided comparator on each `Employee` object.

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**. In this call, `std::ranges::greater` will compare the `int` salaries of `Employees` to determine the sort order.

```

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:
Jones      Amaia      7600
Wu         Joo-won    5000
Makaya     Titania   3587

```

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.

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.

These are **range adaptors**. A view is like a window that enables you to “see” into a range and observe its elements—views do not have their own copy of those elements. Each range adaptor takes a `std::ranges::viewable_range` as an argument and returns a view of that range for use in a pipeline of operations (introduced in Section 6.14). A `viewable_range` is a range that can be “safely converted into a view.”³³ Because views do not own the data that they “see,” a temporary object cannot be converted to a `viewable_range`.

The following table lists many C++20 range adaptors (header `<ranges>`),³⁴ enabling functional-style programming (introduced in Section 6.14). Range adaptors are defined in the namespace `std::ranges::views` and generally accessed via the alias `std::views`.

Range adaptor	Description
<code>filter</code>	Creates a <code>view</code> representing only the <code>range</code> elements for which a <code>predicate</code> returns <code>true</code> .
<code>transform</code>	Creates a <code>view</code> that maps elements to new values.
<code>common</code>	Used to convert a <code>view</code> into a <code>std::ranges::common_range</code> , which enables a <code>range</code> to be used with common-range algorithms that require <code>begin/end iterator</code> pairs of the same type.
<code>all</code>	Creates a <code>view</code> representing all of a <code>range</code> 's elements.
<code>counted</code>	Creates a <code>view</code> of a specified number of elements from either the beginning of a <code>range</code> or from a specific <code>iterator position</code> .
<code>reverse</code>	Creates a <code>view</code> for processing a bidirectional <code>view</code> in reverse order.
<code>drop</code>	Creates a <code>view</code> that ignores the specified number of elements at the beginning of another <code>view</code> .
<code>drop_while</code>	Creates a <code>view</code> that ignores the elements at the beginning of another <code>view</code> as long as a <code>predicate</code> returns <code>true</code> .
<code>take</code>	Creates a <code>view</code> containing the specified number of elements from the beginning of another <code>view</code> .
<code>take_while</code>	Creates a <code>view</code> containing the elements from the beginning of another <code>view</code> as long as a <code>predicate</code> returns <code>true</code> .
<code>join</code>	Creates a <code>view</code> that combines the elements of multiple <code>ranges</code> .
<code>split</code>	Splits a <code>view</code> based on a delimiter. The new <code>view</code> contains a separate <code>view</code> for each <code>subrange</code> .
<code>keys</code>	Creates a <code>view</code> of the <code>keys</code> in key–value pairs, such as those in a <code>map</code> . The <code>keys</code> are the first elements in the <code>pairs</code> .
<code>values</code>	Creates a <code>view</code> of the <code>values</code> in key–value pairs, such as those in a <code>map</code> . The <code>values</code> are the second elements in the <code>pairs</code> .
<code>elements</code>	For a <code>view</code> containing <code>tuples</code> , <code>pairs</code> or <code>arrays</code> , creates a <code>view</code> consisting of elements from a specified index in each object.

33. “`std::ranges::viewable_range`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/ranges/viewable_range.

34. “Ranges Library.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/ranges>.

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 `std::views::iota`. Line 13 defines a `lambda` that returns `true` if its argument is an even integer. We'll use this `lambda` in our pipelines.

```

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

```

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 required 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. Again, `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`, which we'll use `evens` to build several enhanced pipelines.

```

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,³⁵ to process one in a loop or pass one to a standard library algorithm, **you must limit the number of elements the pipeline will produce**; otherwise, your program will contain an infinite loop. One way to do this is to use a range adaptor that limits the number of elements to process. Such adaptors work with

35. Jeff Garland, “Using C++20 Ranges Effectively,” June 18, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=VmWS-9idT3s>.

infinite ranges and fixed-size ranges. For example, line 19 uses the **take range adaptor** to take only the first five values from the **evens pipeline** 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:

- **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**. 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.

```
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
```

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 while **take_while**'s **unary predicate** returns **true**. 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.

```
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
```

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.

```
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
```

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.

```
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
```

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 1,000 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.

```
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
```

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`. It uses the **drop_while range adaptor** to skip even integers at the beginning of the **infinite sequence** that are less than or equal to 1,000. Then it 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.

```
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 lines 51–52. 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::for_each` to display each key–value pair in the `map`.

```
48 // allow std::string object literals
49 using namespace std::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
```

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`.
- `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.

```
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
```

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 70 selects only element 1. In both cases, we pass the pipeline to **std::ranges::copy**, which iterates through the pipelines and displays the results.

```
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
```

14.8 Intro to Parallel Algorithms

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.^{36,37} Computer processing power continues to increase, but hardware vendors now rely on new processor designs, such as **multi-core 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.

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

- 36. Esther Shein, “Moore's Law Turns 55: Is It Still Relevant?” April 17, 2020. Accessed April 18, 2023. <https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant>.
- 37. Nick Heath, “Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow,” September 19, 2018. Accessed April 18, 2023. <https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/>.

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.

Over the years, it has become clear that designing algorithms capable of executing on multiple cores is complex and error-prone. 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 simultaneously.³⁸

In addition, C++17 added seven new parallel algorithms, each of which also has a sequential version:

- `for_each_n`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `reduce`
- `transform_reduce`

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++23.³⁹

Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, will

- overview the parallel algorithms,
- discuss the four “execution policies” 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 vs. running in parallel so you can see the difference in performance.

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. Then, in Chapter 18, we’ll introduce C++20’s new coroutines feature.

38. “General Utilities Library—execution Policies—unsequenced Execution Policy.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq>.

39. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed April 18, 2023. <https://wg21.link/p2214r0>.

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 the algorithms we did not present in this chapter, visit sites such as

<https://en.cppreference.com/w/cpp/algorithm>

<https://docs.microsoft.com/en-us/cpp/standard-library/algorithm>

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.”⁴⁰
- Algorithms that have a parallel version are marked with a superscript “P.”⁴¹
- Algorithms 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**—that is, 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,⁴² making programs that use `rand` less secure. The newer `shuffle` algorithm uses **nondeterministic random-number generation** capabilities.



40. “Constrained Algorithms.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/algorithms/ranges>.

41. “Extensions for Parallelism.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/experimental/parallelism/>. [Though this link contains “experimental,” these parallel versions are officially part of the C++ standard.]

42. “Do Not Use the `rand()` Function for Generating Pseudorandom Numbers.” Last modified April 23, 2021. Accessed April 18, 2023. <https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers>.

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**—that is, algorithms that do not modify the containers they manipulate.

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	

Sorting and related algorithms from header `<algorithm>` (Cont.)

<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

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.^{43,44}

Generalized numeric operations from header `<numeric>`

<code>accumulate</code>	<code>gcd</code> ¹⁷	<code>iota</code> ¹¹
<code>lcm</code> ¹⁷	<code>partial_sum</code>	<code>reduce</code> ^{P17}
<code>adjacent_difference</code> ^P	<code>exclusive_scan</code> ^{P17}	<code>inclusive_scan</code> ^{P17}
<code>inner_product</code> ^P	<code>midpoint</code> ²⁰	<code>transform_reduce</code> ^{P17}
<code>transform_exclusive_scan</code> ^{P17}	<code>transform_inclusive_scan</code> ^{P17}	

Specialized Memory Operations

The following table shows the specialized memory algorithms of the header `<memory>`, which contains features for dynamic memory manipulation operations that are beyond this book's scope. For an overview of the header and these algorithms, see

<https://en.cppreference.com/w/cpp/header/memory>

Specialized Memory Operations

<code>construct_at</code> ^{R20}	<code>destroy</code> ^{R17}
<code>destroy_at</code> ^{R17}	<code>destroy_n</code> ^{R17}
<code>uninitialized_copy</code> ^{PR}	<code>uninitialized_copy_n</code> ^{PR11}
<code>uninitialized_default_construct</code> ^{PR17}	<code>uninitialized_default_construct_n</code> ^{PR17}
<code>uninitialized_fill</code> ^{PR}	<code>uninitialized_fill_n</code> ^{PR}
<code>uninitialized_move</code> ^{PR17}	<code>uninitialized_move_n</code> ^{PR17}
<code>uninitialized_value_construct</code> ^{PR17}	<code>uninitialized_value_construct_n</code> ^{PR17}

43. Christopher Di Bella, “A Concept Design for the Numeric,” August 2, 2019. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1813r0.pdf>.

44. Tristan Brindle, “Numeric Range Algorithms for C++20,” May 19, 2020. Accessed April 18, 2023. <https://tristanbrindle.com/posts/numeric-ranges-for-cpp20>.

14.10 Future Ranges Enhancements

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 various additional ranges features in C++23. Some of these capabilities are under development and can be used now via the open-source project “Ranges for C++23.”⁴⁵ 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-v3.⁴⁶

“A Plan for C++23 Ranges”⁴⁷

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 several 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”⁴⁸ and “Formatting Ranges,”⁴⁹ respectively.
- **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.”⁵⁰ This section also briefly overviews potential issues with producing `std::ranges` overloads of the parallel algorithms. The paper “Introduce Parallelism to the Ranges TS”⁵¹ provides more in-depth discussions of these issues.
- **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.

-
45. Corentin Jabot, “Ranges for C++23.” Accessed April 18, 2023. <https://github.com/cor3ntin/rangesnext>.
 46. Eric Niebler, “range-v3.” Accessed April 18, 2023. <https://github.com/ericniebler/range-v3>.
 47. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” February 18, 2022. Accessed April 18, 2023. <https://wg21.link/p2214r2>.
 48. Corentin Jabot, Eric Niebler and Casey Carter, “ranges::to: A Function to Convert Any Range to a Container,” November 22, 2021. Accessed April 18, 2023. <https://wg21.link/p1206r3>.
 49. Barry Revzin, “Formatting Ranges,” February 19, 2021. Accessed April 18, 2023. <https://wg21.link/p2286>.
 50. Christopher Di Bella, “A Concept Design for the Numeric Algorithms,” August 2, 2019. Accessed April 18, 2023. <http://wg21.link/p1813r0>.
 51. Gordon Brown, Christopher Di Bella, Michael Haidl, Toomas Remmelg, Ruyman Reyes, Michel Steuwer and Michael Wong, “P0836R1 Introduce Parallelism to the Ranges TS,” May 7, 2018. Accessed April 18, 2023. <https://wg21.link/p0836r1>.

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 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 17, these will enable you to take advantage of your computer's multi-core hardware to enhance program performance. In Chapter 17, 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 proposed updates.

In the next chapter, we'll build a simple container, iterators and a simple algorithm using custom templates, using C++20 concepts in our templates as appropriate.

Self-Review Exercises

14.1 State whether each of the following is *true* or *false*. If *false*, explain why.

- Standard library algorithms are encapsulated as member functions within each container class.
- When using the `remove` algorithm on a container, the algorithm does not decrease the size of the container from which elements are being removed.
- One disadvantage of using standard library algorithms is that they depend on the implementation details of the containers on which they operate.

- d) The `remove_if` algorithm does not modify the number of elements in the container, but it does move to the beginning of the container all elements that are not removed.
- e) The `find_if_not` algorithm locates all the values in the range for which the specified unary predicate function returns `false`.
- f) Use the `set_union` algorithm to create a set of all the elements that are in either or both of two sorted sets—both sets of values must be in ascending order.
- 14.2** Fill in the blanks in each of the following statements:
- Standard library algorithms operate on container elements indirectly via _____.
 - The `sort` algorithm requires a(n) _____ iterator.
 - Algorithms _____ and _____ set every element in a range of container elements to a specific value.
 - The _____ algorithm compares two sequences of values for equality.
 - The _____ algorithm locates a range's smallest and largest elements.
 - A `back_inserter` calls the container's _____ function to insert an element at the end of the container. If an element is inserted into a container that has no more space available, the container grows in size.
 - Any standard library algorithm that can receive a function pointer can also receive an object of a class that defines an overloaded `operator()` function, provided that the overloaded operator meets the requirements of the algorithm. An object of such a class is known as a(n) _____ and can be used syntactically and semantically like a function or function pointer.
- 14.3** Write a statement to perform each of the following tasks:
- Use the `fill` algorithm to fill an array of `strings` named `items` with "hello".
 - Function `nextInt` returns the next `int` value in sequence starting with 0 the first time it's called. Use the `generate` algorithm and the `nextInt` function to fill the array of `ints` named `integers`.
 - Use the `equal` algorithm to compare two lists (`strings1` and `strings2`) for equality. Store the result in `bool` variable `result`.
 - Use the `remove_if` algorithm to remove from the `vector` of `strings` named `colors` all of the strings that start with "bl". Function `startsWithBL` returns `true` if its argument `string` starts with "bl". Store the iterator that the algorithm returns in `newEnd`.
 - Use the `replace_if` algorithm to replace with 0 all elements with values greater than 100 in the array of `ints` named `values`. Function `greaterThan100` returns `true` if its argument is greater than 100.
 - Use the `minmax_element` algorithm to find the smallest and largest values in an array of `doubles` named `temperatures`. Store the returned pair of iterators in `result`.
 - Use the `sort` algorithm to sort the array of `strings` named `colors`.
 - Use the `reverse` algorithm to reverse the order of the elements in the array of `strings` named `colors`.
 - Use the `merge` algorithm to merge the contents of the two sorted arrays named `values1` and `values2` into a third array named `results`.
 - Write a lambda expression that returns the square of its `int` argument and assign the lambda expression to variable `squareInt`.

Answers to Self-Review Exercises

14.1 a) False. Standard library algorithms are not member functions. They operate indirectly on containers via iterators. b) True. c) False. Standard library algorithms do not depend on the implementation details of the containers on which they operate. d) True. e) False. It locates only the first value in the range for which the specified unary predicate function returns `false`. f) True.

14.2 a) iterators. b) random-access. c) `fill`, `fill_n`. d) `equal`. e) `minmax_element`. f) `push_back`. g) function object.

14.3

- a) `std::ranges::fill(items, "hello");`
- b) `std::ranges::generate(integers, nextInt);`
- c) `bool result{std::ranges::equal(strings1, strings2)};`
- d) `auto newEnd {std::ranges::remove_if(colors, startsWithBL)};`
- e) `std::ranges::replace_if(values, greaterThan100, 0);`
- f) `auto result {std::ranges::minmax_element(temperatures)};`
- g) `std::ranges::sort(colors);`
- h) `std::ranges::reverse(colors);`
- i) `std::ranges::merge(values1, values2, results.begin());`
- j) `auto squareInt{[](int i) {return i * i;}};`

Exercises

14.4 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Because standard library algorithms process containers directly, one algorithm can often be used with many different containers.
- b) Use the `for_each` algorithm to apply a general function to every element in a range; `for_each` does not modify the sequence.
- c) By default, the `sort` algorithm arranges the elements in a range in ascending order.
- d) Use the `merge` algorithm to form a new sequence by placing the second sequence after the first.
- e) Use the `set_intersection` algorithm to find the elements from a first set of sorted values that are not in a second set of sorted values (both sets of values must be in ascending order).
- f) Algorithms `lower_bound`, `upper_bound` and `equal_range` are often used to locate insertion points in sorted sequences.
- g) Lambda expressions can also be used where function pointers and function objects are used in algorithms.
- h) Lambda expressions can “capture” (by value or by reference) local variables of the enclosing function then manipulate these variables in the lambda’s body.

14.5 Fill in the blanks in each of the following statements:

- a) As long as a container’s _____ satisfy the requirements of an algorithm, the algorithm can work on the container.
- b) Algorithms `generate` and `generate_n` use a(n) _____ function to create values for every element in a range of container elements. That type of function takes no arguments and returns a value that can be placed in an element of the container.

- c) Use the _____ algorithm (header <numeric>) to sum the values in a range.
- d) Use the _____ algorithm to apply a general function to every element in a range when you need to modify those elements.
- e) The `binary_search` algorithm requires that the sequence of values must be _____.
- f) Use the function `iter_swap` to exchange the elements that are pointed to by two _____ iterators and exchanges the values in those elements.
- g) The two-argument `minmax` algorithm returns a(n) _____ in which the smaller item is stored in `first`, and the larger item is stored in `second`.
- h) _____ algorithms modify the containers they operate on.

14.6 List several advantages function objects provide over function pointers.

14.7 What happens when you apply the `unique` algorithm to a sorted sequence of elements in a range?

14.8 (*Duplicate Elimination*) Read 20 integers into an array. Next, sort the array, then use the `unique` algorithm to reduce the array to the unique values entered by the user. Use the `copy` algorithm to display the unique values.

14.9 (*Duplicate Elimination*) Modify Exercise 14.8 to use the `unique_copy` algorithm. Insert the unique values into a vector that's initially empty. Use a `back_inserter` to grow the vector as new items are added. Use the `copy` algorithm to display the unique values.

14.10 (*Reading Data from a File*) Use an `istream_iterator<int>`, the `copy` algorithm and a `back_inserter` to read the contents of a text file that contains `int` values separated by whitespace. Place the `int` values into a vector of `ints`. The first argument to the `copy` algorithm should be the `istream_iterator<int>` object that's associated with the text file's `ifstream` object. The second argument should be an `istream_iterator<int>` object that's initialized using the class template `istream_iterator`'s default constructor—the resulting object can be used as an “end” iterator. After reading the file's contents, display the contents of the resulting vector.

14.11 (*Merging Ordered Lists*) Write a program that uses standard library algorithms to merge two ordered lists of strings into a single ordered list of strings, then displays the resulting list.

14.12 (*Palindrome Tester*) A palindrome is a string that is spelled the same way forward and backward. Examples of palindromes include “radar” and “able was i ere i saw elba.” Write a function `palindromeTester` that uses the `reverse` algorithm on a copy of a string, then compares the original string and the reversed string to determine whether the original string is a palindrome. Like the standard library containers, `string` objects are ranges. Assume that the original string contains all lowercase letters and does not contain any punctuation. Use function `palindromeTester` in a program.

14.13 (*Enhanced Palindrome Tester*) Enhance Exercise 14.12's `palindromeTester` function to allow strings containing uppercase and lowercase letters and punctuation. Before testing whether the original string is a palindrome, function `palindromeTester` should convert the string to lowercase letters and eliminate any punctuation. For simplicity, assume the only punctuation characters can be

. , ! ; : ()

You can use the `copy_if` algorithm and a `back_inserter` to make a copy of the original `string`, eliminate the punctuation characters and place the characters into a new `string` object.

14.14 Explain why using the “weakest iterator” that yields acceptable performance helps produce maximally reusable components.

14.15 (Functional-Style Programming: Summing the Cubes of the Odd Integers from 1 through 9) Use the techniques from Fig. 14.18 to produce the cubes of the odd integers from 1 through 9, then calculate and display their sum.

14.16 (Functional-Style Programming: Filtering and Sorting) Generate 50 random numbers in the range 1 to 999, then use the techniques from Fig. 14.18 to filter the results, keeping only the odd numbers. Display the results in sorted order.

14.17 (Functional-Style Programming: array of Invoices) Use the `Invoice` class provided in the `exercises` folder with this chapter’s examples to create an array of `Invoice` objects. Use the sample data shown in the following table:

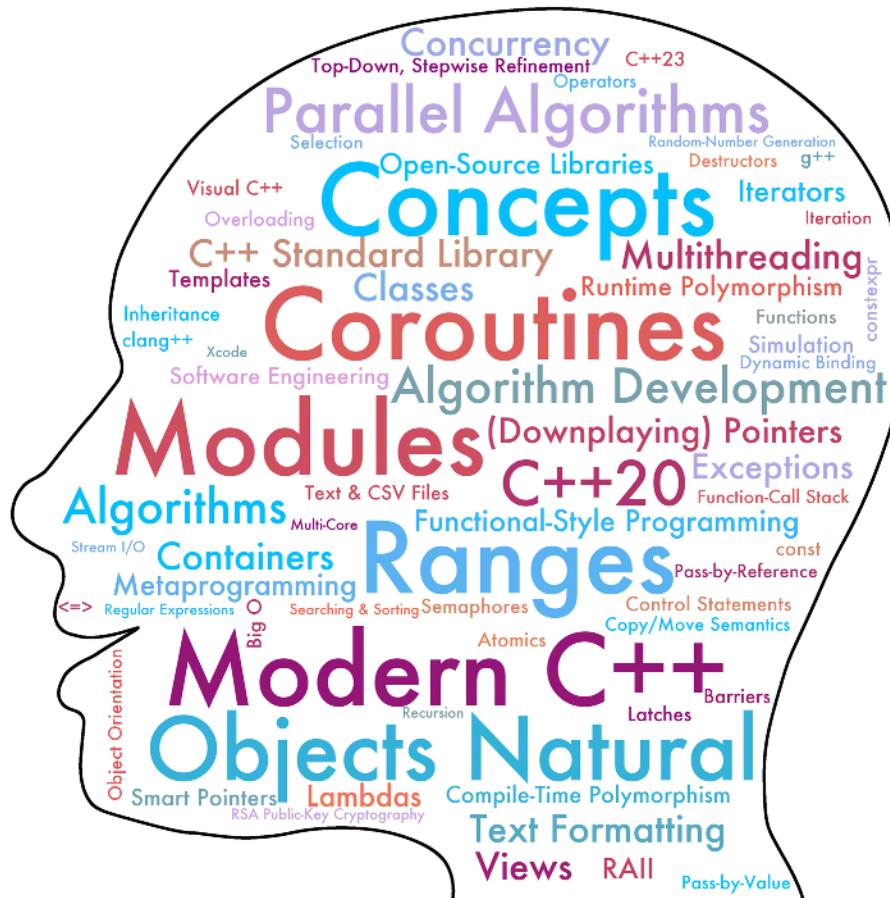
Part number	Part description	Quantity	Price
83	Electric sander	7	57.98
24	Power saw	18	99.99
7	Sledge hammer	11	21.50
77	Hammer	76	11.99
39	Lawn mower	3	79.50
68	Screwdriver	106	6.99
56	Jig saw	21	11.00
3	Wrench	34	7.50

An `Invoice` contains a `partNumber` (`string`), a `partDescription` (`string`), a `quantity` of the item being purchased (`int`) and a `pricePerItem` (`double`), and corresponding `get` member functions. Using the techniques from Sections 14.6–14.7, perform the following queries on the array of `Invoice` objects and display the results:

- Sort the `Invoice` objects by `partDescription`, then display the results.
- Sort the `Invoice` objects by `pricePerItem`, then display the results.
- Map each `Invoice` to its `partDescription` and `quantity`, sort the results by `quantity`, then display the results.
- Map each `Invoice` to its `partDescription` and the value of the `Invoice` (that is, `quantity * pricePerItem`). Order the results by `Invoice` value.
- Modify Part (d) to select the `Invoice` values in the range \$200 to \$500.

14.18 (Functional-Style Programming: Calculating Employee Average Salaries by Department) Create an `Employee` aggregate type containing data members `firstName` (a `string`), `lastName` (a `string`), `salary` (a `double`) and `department` (a `string`). Create a vector of `Employees` with three `Employees` in each of the departments "IT", "Sales" and "Marketing". Use the techniques from Sections 14.6–14.7 to calculate and display the average `Employee` salary by department.

Templates, C++20 Concepts and Metaprogramming



Objectives

In this chapter, you'll:

- Appreciate the importance of generic programming.
- Create custom class templates.
- Understand compile-time vs. runtime polymorphism.
- Understand templates vs. template instantiations.
- Use C++20 abbreviated function templates and templated lambdas.
- Use C++20 Concepts to constrain template parameters and overload function templates based on 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 class `MyArray` class as a container class template with custom iterators.
- Use non-type template parameters and default template arguments.
- Create variadic templates with any number of parameters and apply binary operators to them via fold expressions.
- Use compile-time template metaprogramming 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	Unconstrained Function Template <code>multiply</code>	
15.4.2	Constrained Function Template with a C++20 Concepts <code>requires</code> 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	<code>requires</code> 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 <code>static_assert</code>	
15.8	Creating a Custom Algorithm	
15.9	Creating a Custom Container and Iterators	
15.9.1	Class Template <code>ConstIterator</code>	
15.9.2	Class Template <code>Iterator</code>	
15.9.3	Class Template <code>MyArray</code>	
15.9.4	MyArray Deduction Guide for Braced Initialization	
15.9.5	Using <code>MyArray</code> with <code>std::ranges</code> Algorithms	
15.10	Default Arguments for Template Type Parameters	
15.11	Variable Templates	
15.12	Variadic Templates and Fold Expressions	
15.12.1	<code>tuple</code> Variadic Class Template	
15.12.2	Variadic Function Templates and an Intro to 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 <code>constexpr if</code>	
15.13.4	Type Metafunctions	
15.14	Wrap-Up Exercises	

15.1 Introduction

Generic programming with templates has been in C++ since the 1998 C++ standard was released¹ and has increased in importance with each new release. A modern C++ theme is doing more at compile-time for better type checking and runtime performance.^{2,3,4} You've already used templates extensively. As you'll see, templates and template metaprogramming are the keys to powerful compile-time operations. In this chapter, we'll take a deeper look at 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 20 chapters.

- 1. "History of C++." Accessed April 18, 2023. <https://wwwcplusplus.com/info/history/>.
- 2. C++ Core Guidelines, "Per: Performance." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance>.
- 3. "Big Picture Issues—What's the Big Deal with Generic Programming?" Accessed April 18, 2023. <https://isocpp.org/wiki/faq/big-picture#generic-paradigm>.
- 4. "Compile Time vs. Run Time Polymorphism in C++: Advantages/Disadvantages." Accessed April 18, 2023. <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— <code>string</code> is an alias for <code>basic_string<char></code>).
Chapter 5	Section 5.8: <code>uniform_int_distribution</code> class template for random-number generation. Section 5.15: 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	Objects Natural Case Study Exercise: Serialization with JSON— <code>cereal</code> library function templates for serializing and deserializing data using JavaScript Object Notation (JSON).
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 19	Stream I/O classes (which are class templates).
Chapter 20	Section 20.3: Runtime polymorphism with the <code>std::variant</code> class template and the <code>std::visit</code> function template. Section 20.2: <code>shared_ptr</code> and <code>weak_ptr</code> smart-pointer class templates.

Custom Templates

Section 5.15 showed that the compiler uses a function template to generate overloaded functions—**instantiating the function template**. Similarly, the compiler uses class templates to generate related classes—this is known as **instantiating the class template**. Instantiating 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

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* instantiating templates’ bodies, often resulting in fewer more-precise error messages.

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 define **custom iterators** to make `MyArray` objects compatible with many standard library algorithms. We’ll also define a **custom algorithm** that can process `MyArray` elements and standard library container class objects.

Variadic Templates and Fold Expressions

We’ll build a variadic function template that receives a variable number of parameters. We’ll also introduce **fold expressions**, which enable you to conveniently apply an operation to all the items passed to a variadic template.

Template Metaprogramming

The C++ Core Guidelines define template metaprogramming (TMP) as “**creating programs that compose 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**.

Before C++20, many C++ programmers viewed template metaprogramming as too complex. C++20 Concepts make aspects of it friendlier and more accessible. Nevertheless,

5. Marius Bancila, “Concepts Versus SFINAE-Based Constraints.” Accessed April 18, 2023. <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 April 18, 2023. <https://en.cppreference.com/w/cpp/language/constraints>.
8. “Concepts Library.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/concepts>.
9. C++ Core Guidelines, “T: Templates and Generic Programming.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates>.

Google's C++ Style Guide 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 very hard to maintain."¹¹ These guidelines could be updated when developers gain more experience using C++20 Concepts and future C++ enhancements.



15.2 Custom Class Templates and Compile-Time Polymorphism

A template enables **compile-time** (or **static**) **polymorphism**^{12,13} by specifying capabilities generically, then letting the compiler instantiate the template, generating type-specific code specializations on demand. Class templates are called **parameterized types** because they require one or more **parameters** to tell the compiler how to customize a 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 to instantiate the template. Then, the compiler writes only those specializations for you. One Stack class template, for example, could become the basis for creating many 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 usually occur when you instantiate 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. A stack is simply a last-in, first-out (LIFO) data structure. Class templates encourage software reuse by enabling the compiler to instantiate many type-specific class-template specializations from a single class template—as you saw with the **stack** container adaptor in Section 13.12.1. Here (Fig. 15.1), we define a stack generically, then instantiate and



-
10. "Google C++ Style Guide—Template Metaprogramming." Accessed April 18, 2023. https://google.github.io/styleguide/cppguide.html#Template_metaprogramming.
 11. C++ Core Guidelines, "T.120: Use Template Metaprogramming Only When You Really Need To." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta>.
 12. "C++—Static Polymorphism." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.
 13. Kateryna Bondarenko, "Static Polymorphism in C++," May 6, 2019. Accessed April 18, 2023. <https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ae27a945b>.

use type-specific stacks (Fig. 15.2). Figure 15.1’s `Stack` class-template definition looks like a conventional class definition, with a few key differences.

```

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 Stack
10    const T& top() const {return stack.front();}
11
12    // push an element onto Stack
13    void push(const T& pushValue) {stack.push_front(pushValue);}
14
15    // pop an element from Stack
16    void pop() {stack.pop_front();}
17
18    // determine whether Stack is empty
19    bool isEmpty() const {return stack.empty();}
20
21    // return size of Stack
22    size_t size() const {return stack.size();}
23 private:
24    std::deque<T> stack{}; // internal representation of Stack
25 };

```

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 representing 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. (There are other cases in which **typename** must be used rather than **class**.^{14,15}) The **type parameter** **T** is a **placeholder** for the `Stack`’s **element type**. Type parameter names can be any valid identifier but must be unique inside a template definition. **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++, 3/e*. Pearson Education, Inc., 2005.

15. John Lakos, “1.3 Declarations, Definitions, and Linkage,” *Large-Scale C++*. Addison-Wesley, 2020. Footnotes 56 and 57.

- declaring a member function return type (line 10),
- declaring a member function parameter (line 13) and
- declaring a variable (line 24).

When you instantiate the class template, the compiler associates the type parameter with a **type argument**. 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.**^{16,17}

Another difference between the Stack class template 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 instantiated with new type arguments to 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

Section 13.12.1 showed that the standard library's **stack adaptor 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** can 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 adaptor**¹⁸ 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 or moving the old elements into that new block**.
- A **deque** is typically implemented as a list of fixed-size, built-in arrays—with new ones added as necessary. **No existing elements are copied or moved when new items are added to a deque's front or back.**



For these reasons, we use a deque (line 24) as our Stack class's underlying container.

Class Template Stack<T>'s Member Functions

A class template's member-function definitions behave like function templates. **When you define them within the class template's body, you do not precede them with a template header.** They still use the template parameter T to represent the element type. The Stack class template does not define its own constructors—the class's **default constructor invokes the deque data member's default constructor**.

Stack (Fig. 15.1) provides the following member functions:

- **top** (line 10) returns a **const** reference to the Stack's top element without removing it. You could overload **top** with a non-**const** version as well.

16. Andreas Fertig, "Back to Basics: Templates (Part 1 of 2)," September 25, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=VNJ4wiuxJM4>.

17. "13.9.2 Implicit Instantiation [temp.inst]." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/temp.inst#4>.

18. "std::stack." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/container/stack>.

- **push** (line 13) places a new element on the top of the Stack.
- **pop** (line 16) removes the Stack’s top element.
- **isEmpty** (line 19) returns true if the Stack is empty; otherwise, it returns false.
- **size** (line 22) 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 the stack’s elements. The output shows that the values indeed pop off in **last-in, first-out** order. When **doubleStack** is empty, the **pop** loop terminates.

```

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

```

Fig. 15.2 | Stack class template test program. (Part 1 of 2.)

```

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.

Fig. 15.2 | Stack class template test program. (Part 2 of 2.)

Line 31 instantiates `intStack` as a `Stack<int>` (pronounced “Stack of int”). Lines 38–41 repeatedly call `push` (line 39) to place values onto `intStack`. Then, lines 46–49 repeatedly call `isEmpty`, `top` and `pop` to remove values from `intStack` until it’s empty. 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 instantiations generated by the Clang C++ compiler.¹⁹

19. The site requires all the code to be pasted into its code pane. To try this with our example, remove the `#include` for `stack.h` in the `main` program and paste the `Stack` class template’s code above `main`. Also, remove the `#pragma once` directive.

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

```
template<typename T>
void Stack<T>::pop() {stack.pop_front();}
```

`Stack<T>::` indicates that `pop` is in class template `Stack<T>`'s scope. Standard library class templates define some member functions inside and some outside the class-template bodies.

15.3 C++20 Function Template Enhancements

In addition to concepts, C++20 added abbreviated function templates and templated lambda expressions.

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:

```
template <typename T>
void printContainer(const T& items) {
    for (const auto& item : items) {
        std::cout << item << " ";
    }
}
```

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) by using the `auto` keyword as the parameter type (line 10).

```

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's elements separated by spaces
10 void printContainer(const auto& items) {
11     for (const auto& item : items) {
12         std::cout << item << " ";
13     }
14 }
```

Fig. 15.3 | Abbreviated function template. (Part I of 2.)

```

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

```

Fig. 15.3 | Abbreviated function template. (Part 2 of 2.)

Lines 19–20 define an array of `ints` and a vector of `strings` (line 20), which we initialized with `string` object literals. The compiler uses CTAD to deduce their element types from each initializer list (and the array's size as well). In the `printContainer` calls at lines 23 and 25, the compiler infers the parameter type from the containers passed as arguments and generates appropriate template instantiations for each.

Sometimes Traditional Function Template Syntax Is Required

The abbreviated function template syntax is similar to regular function definitions but is not always appropriate. Consider the first two lines of Section 5.15's function template `maximum`, which received three parameters of the same type:

```

template <typename T>
T maximum(T value1, T value2, T value3) {

```

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

```

auto maximum(auto value1, auto value2, auto value3) {

```

the compiler independently infers each `auto` parameter's type based on the corresponding argument. So, `maximum` might receive arguments of three different types.

Using `printContainer` with an Incompatible Type

When the compiler attempts to instantiate the `printContainer` function template, errors  Err 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.

Such errors are often confusing, as they mention `printContainer` internal implementation details that the caller does not need to know. If the argument is incompatible, it would be better for client-code programmers if the error message(s) simply stated why. In Sections 15.4 and 15.6, we'll show that C++20 Concepts can constrain the types passed to a function template and prevent the compiler from attempting to instantiate the tem-

plate. As you'll see, you'll also receive error messages that generally are easier to understand.

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:

```
[](auto total, auto value) {return total + value * value;}
```

 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:

```
[]<typename T>(T total, T value) {return total + value * value;}
```

The template parameter list is placed between the lambda's introducer and parameter list. When you use one type parameter (`T`) to declare both lambda parameters, the compiler requires both arguments to have the same type; otherwise, a compilation error occurs.


15.4 C++20 Concepts: A First Look

 Concepts (briefly introduced in Section 14.2) simplify generic programming. C++ experts say that “Concepts are a revolutionary approach for writing templates”²⁰ and that “C++20 creates a paradigm shift in the way we use metaprogramming.”²¹ 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 work-arounds (e.g., traits classes) and low-level techniques (e.g., `enable_if`-based overloading) feel like error-prone and tedious assembly programming.”²²

As you'll see, concepts 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 (e.g., “Is the type an integer type?”) and
- test whether types support various operations (e.g., “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 defines a type's requirements or a relationship between types.²³ Concepts can be applied to any template parameter and to any use of `auto`.²⁴ We'll discuss concepts in more depth in Section 15.6. Initially, we'll focus on using concepts to constrain a function template's type parameters.

20. Bartłomiej Filipek, “C++20 Concepts—A Quick Introduction,” May 5, 2021. Accessed April 18, 2023. <https://www.cppstories.com/2021/concepts-intro/>.

21. Inbal Levi, “Exploration of C++20 Meta Programming,” September 29, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=XgrjybkaIV8>.

22. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces,” January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.

23. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces,” January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.

24. “Placeholder Type Specifiers.” Accessed April 18, 2023. <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.”²⁵

Traditionally, **template requirements were implicit**, based on how the template used its arguments in operator expressions, function calls, etc.²⁶ This was the case in Fig. 15.3’s abbreviated function template `printContainer`. The function definition did not indicate that the argument must be iterable with the range-based `for` statement or that the element type must support the `<<` operator for output. 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 to instantiate the template to “see” that a type did not support the template’s implicit requirements. This led to many, often cryptic, compilation errors.

Concepts specify template requirements explicitly in code. They enable the compiler to determine that a type is incompatible with a template *before* instantiating it—leading to fewer, more precise error messages and potential compile-time performance improvements.²⁷



Concepts also enable you to overload function templates with the same signature based on each function template’s requirements.²⁸ 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**²⁹ and a technique called **argument-dependent lookup (ADL)**^{30,31,32,33} to locate all the function definitions that might satisfy the function call. Together, these are known as the **overload set**. This process often includes instantiating 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

-
25. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3 Using Concepts,” January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.
 26. Hendrik Niemeyer, “An Introduction to C++20’s Concepts,” July 25, 2020. Accessed April 18, 2023. https://www.youtube.com/watch?v=N_kPd20K1L8.
 27. The error messages’ quality and quantity still vary considerably among compilers.
 28. This enables even a class’s no-argument constructor to be overloaded with specific constraints.
 29. “Overload Resolution.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/overload_resolution.
 30. C++ Standard, “6.5.4 Argument-Dependent Name Lookup [basic.lookup.argdep].” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/basic.lookup.argdep#:lookup,argument-dependent>.
 31. “Argument-Dependent Lookup.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/adl>.
 32. Inbal Levi, “Exploration of C++20 Metaprogramming,” September 29, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=XgrjybKaIV8>.
 33. Arthur O’Dwyer, “What Is ADL?” April 26, 2019. Accessed April 18, 2023. <https://quuxplu-sone.github.io/blog/2019/04/26/what-is-adl/>.

compiler substitutes the call's argument types into the function template's declaration to check whether it is a viable match and, if so, whether it is the best match. If it's the best match, the compiler will instantiate the template's definition and determine whether the argument types support the operations used in the function template's body.

Consider the **unconstrained function template** `multiply` (lines 5–6 of Fig. 15.4), which receives **two values of the same type** (`T`) and returns their product.

```

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) {return first * second;}
7
8 int main() {
9     std::cout << "Product of 5 and 3: " << multiply(5, 3)
10    << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
11 }
```

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 (line 6) 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. Thus, **the arguments' type must support copying or moving**.
- The arguments are multiplied, 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 instantiates `multiply` for a given type and determines that the arguments are incompatible with the template's implicit requirements, it eliminates the **function from the overload set**. In this example, lines 9 and 10 call `multiply` with two `int` values and two `double` values, respectively. All numeric types in C++ support this template's **implicit requirements**, so the compiler can instantiate `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`

What if we pass to `multiply` arguments that do not support the template's **implicit requirements** and there is no other function declaration that better matches the function call? The compiler will generate error messages. For example, the following code attempts to calculate the product of two `strings`:³⁴



34. For your convenience, this code is provided in the source-code file as comments at the end of `main`.

```
std::string s1{"hi"};
std::string s2{"bye"};
auto result{multiply(s1, s2)}; // string does not have * operator
```

Class `string` 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

```
fig15_04.cpp:6:45: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
T multiply(T first, T second) {return first * second;}
                ~~~~~ ^ ~~~~~

fig15_04.cpp:14:16: note: in instantiation of function template
specialization 'multiply<std::basic_string<char>>' requested here
    auto result{multiply(s1, s2)}; // string does not have * operator
    ^
1 error generated.
```

GNU g++ produces

```
fig15_04.cpp: In instantiation of 'T multiply(T, T) [with T =
std::__cxx11::basic_string<char>]':
fig15_04.cpp:14:24:   required from here
fig15_04.cpp:6:45: error: no match for 'operator*' (operand types are
'std::__cxx11::basic_string<char>' and 'std::__cxx11::basic_string<char>')
  6 | T multiply(T first, T second) {return first * second;}
     |           ~~~~~^~~~~~
```

Visual C++ produces

```
1>fig15_04.cpp
1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(6,45): error C2676: bi-
nary '*': 'T' does not define this operator or a conversion to a type accept-
able to the predefined operator
1>          with
1>          [
1>              T=std::string
1>          ]

1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(14): message : see ref-
erence to function template instantiation 'T multiply<std::string>(T,T)' be-
ing compiled
1>          with
1>          [
1>              T=std::string
1>          ]

1>Done building project "concurrency_test.vcxproj" -- FAILED.
```

These 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 containing only the following two simple statements that attempt to sort a `std::list`:

```
std::list integers{10, 2, 33, 4, 7, 1, 80};
std::sort(integers.begin(), integers.end());
```

A `std::list` has **bidirectional iterators**, but `std::sort` requires **random-access iterators**. One of our preferred compilers generated more than 1,000 lines of error messages for the `std::sort` call! Simply switching to the concept-constrained `std::ranges::sort` algorithm produces far fewer messages and indicates that **random-access iterators** are required.

15.4.2 Constrained Function Template with a C++20 Concepts **requires** Clause

 C++20 Concepts enable you to specify **constraints**. Each is a **compile-time predicate expression** that evaluates to `true` or `false`.³⁵ The compiler uses constraints to check type requirements before instantiating templates. The C++ Core Guidelines recommend

- specifying concepts for every template parameter³⁶ and
- using the standard's predefined concepts if possible.³⁷

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 instantiates 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, more precise error messages.

requires Clause

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 combined with the logical `&&` and `||` operators.

```

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) {return first * second;}
```

Fig. 15.5 | Constrained `multiply` function template that allows only integers and floating-point values. The output shows the `g++` compiler's error messages. (Part 1 of 2.)

-
35. C++ Standard, “13.5 Template Constraints.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/temp.constr>.
 36. C++ Core Guidelines, “T.10: Specify Concepts for All Template Arguments.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts>.
 37. C++ Core Guidelines, “T.11: Whenever Possible Use Standard Concepts.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts>.

```
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
14
15     std::string s1{"hi"};
16     std::string s2{"bye"};
17     auto result{multiply(s1, s2)};
18 }
```

```
fig15_05.cpp: In function 'int main()':
fig15_05.cpp:17:24: error: no matching function for call to
'multiply(std::string&, std::string&)'
    17 |     auto result{multiply(s1, s2)};
    |             ~~~~~~^~~~~~
fig15_05.cpp:9:3: note: candidate: 'template<class T> requires (integral<T>
|| (floating_point<T>) T multiply(T, T)'
    9 | T multiply(T first, T second) {return first * second;}
    | ~~~~~~
fig15_05.cpp:9:3: note: template argument deduction/substitution failed:
fig15_05.cpp:9:3: note: constraints not satisfied
fig15_05.cpp: In substitution of 'template<class T> requires (integral<T>
|| (floating_point<T>) T multiply(T, T) [with T = std::__cxx11::ba-
sic_string<char>]':
fig15_05.cpp:17:24: 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>
  | ~~~~~~^~~~~~
```

Fig. 15.5 | Constrained multiply function template that allows only integers and floating-point values. The output shows the q++ compiler's error messages. (Part 2 of 2.)

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.
 - `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 these types' values work with the `*` operator in line 9. If neither of the preceding constraints is satisfied, the type arguments are incompatible with the `multiply` function template, and the compiler will not instantiate the template. If no other functions match the call, the compiler generates error messages.

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.³⁸



38. C++ Standard, “Index of Library Concepts.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/conceptindex>.

Section 15.4.3 lists the predefined concept categories, the concepts in each and the headers that define them.

Disjunctions and Conjunctions

In line 8, the logical OR (`||`) operator forms a **disjunction**. At least one operand must be true for the compiler to instantiate 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 instantiate the template. Disjunctions and conjunctions use **short-circuit evaluation** (Section 4.11.3).

Calling `multiply` with Arguments That Satisfy Its Constraints

Lines 12 and 13 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 7–9. It will check the arguments' types to determine whether they satisfy either concept in line 8's **requires clause**. Both arguments in each call satisfy one of the **disjunction**'s requirements, so the compiler will instantiate the template for type `int` in line 12 and type `double` in line 13. 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 instantiate the template and will generate errors.

Calling `multiply` with Arguments That Do Not Satisfy Its Constraints

Line 17 calls `multiply` with two `string` arguments. When the compiler looks for function definitions that match this call, it will encounter only the `multiply` function template. Next, it will check the arguments' types to determine whether they satisfy either concept listed in line 8's **requires clause**. The **`string` type does not satisfy either**. Also, no other `multiply` functions can receive two `string` arguments, so the compiler generates error messages. Figure 15.5 shows `g++`'s error 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. For this example, Visual C++ produced the simplest error messages of our three preferred compilers:

```
1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,16): error
C2672: 'multiply': no matching overloaded function found

1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,31): error
C7602: 'multiply': the associated constraints are not satisfied

1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(9): message :
see declaration of 'multiply'
```

15.4.3 C++20 Predefined Concepts

Concepts  The C++ standard's "Index of library concepts"³⁹ alphabetically lists the concepts defined in the standard. The following table lists all the standard concepts by header—`<con-`

39. C++ Standard, "Index of Library Concepts." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/conceptindex>.

`<concepts>`, `<iterator>`, `<ranges>`, `<compare>` and `<random>`. We divided the `<concepts>` header's 31 concepts into 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.

74 predefined C++20 Concepts

`<concepts>` header core language concepts

<code>assignable_from</code>	<code>default_initializable</code>	<code>same_as</code>
<code>common_reference_with</code>	<code>derived_from</code>	<code>signed_integral</code>
<code>common_with</code>	<code>destructible</code>	<code>swappable</code>
<code>constructible_from</code>	<code>floating_point</code>	<code>swappable_with</code>
<code>convertible_to</code>	<code>integral</code>	<code>unsigned_integral</code>
<code>copy_constructible</code>	<code>move_constructible</code>	

`<concepts>` header comparison concepts

<code>equality_comparable</code>	<code>totally_ordered</code>	<code>totally_ordered_with</code>
<code>equality_comparable_with</code>		

`<concepts>` header object concepts

<code>copyable</code>	<code>regular</code>	<code>semiregular</code>
<code>movable</code>		

`<concepts>` header callable concepts

<code>equivalence_relation</code>	<code>predicate</code>	<code>relation</code>
<code>invocable</code>	<code>regular_invocable</code>	<code>strict_weak_order</code>

`<iterator>` header concepts

<code>bidirectional_iterator</code>	<code>indirectly_copyable_storable</code>	<code>input_or_output_iterator</code>
<code>contiguous_iterator</code>	<code>indirectly_movable</code>	<code>mergeable</code>
<code>forward_iterator</code>	<code>indirectly_movable_storable</code>	<code>output_iterator</code>
<code>incrementable</code>	<code>indirectly_readable</code>	<code>permutable</code>
<code>indirect_binary_predicate</code>	<code>indirectly_regular_unary_</code>	<code>random_access_iterator</code>
<code>indirect_equivalence_relation</code>	<code>invocable</code>	<code>sentinel_for</code>
<code>indirect_strict_weak_order</code>	<code>indirectly_swappable</code>	<code>sized_sentinel_for</code>
<code>indirect Unary_predicate</code>	<code>indirectly_unary_invocable</code>	<code>sortable</code>
<code>indirectly_comparable</code>	<code>indirectly_writable</code>	<code>weakly_incrementable</code>
<code>indirectly_copyable</code>	<code>input_iterator</code>	

`<ranges>` header concepts

<code>bidirectional_range</code>	<code>forward_range</code>	<code>random_access_range</code>
<code>borrowed_range</code>	<code>input_range</code>	<code>sized_range</code>
<code>common_range</code>	<code>output_range</code>	<code>view_range</code>
<code>contiguous_range</code>	<code>range</code>	<code>viewable_range</code>

`<compare>` header concepts

<code>three_way_comparable</code>	<code>three_way_comparable_with</code>
-----------------------------------	--

`<random>` header concept

<code>uniform_random_bit_generator</code>

15.5 Type Traits

The `<type_traits>` header⁴⁰ is used for

- 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`)

and use different template code to handle each case. Each subsequent C++ version has added more type traits, and a few have been deprecated or removed.

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 instantiation*, often leading to many cryptic error messages. On the other hand, the compiler tests concepts *before instantiating templates*, typically resulting in fewer, more precise error messages than when you use type traits in unconstrained templates.

C++20 Predefined Concepts Often Use Type Traits

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::nothrow_destructible`.

Demonstrating Type Traits

Figure 15.6 shows type traits corresponding to the concepts in Fig. 15.5.

```

1 // fig15_06.cpp
2 // Using type traits to test whether types are
3 // integral types, floating-point types or arithmetic types.
4 #include <format>
5 #include <iostream>
6 #include <string>
7 #include <type_traits>

```

Fig. 15.6 | Using type traits to test whether types are integral types, floating-point types or arithmetic types. (Part 1 of 2.)

40. “Standard Library Header `<type_traits>`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/header/type_traits.

```

8
9 int main() {
10    std::cout << std::format("{}\n{}{}\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 << std::format("{}\n{}{}\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 << std::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}\n",
33        "CHECK WITH TYPE TRAITS WHETHER TYPES CAN BE USED IN ARITHMETIC",
34        "std::is_arithmetic<int>::value: ", std::is_arithmetic<int>::value,
35        "std::is_arithmetic_v<int>: ", std::is_arithmetic_v<int>,
36        "std::is_arithmetic_v<double>: ", std::is_arithmetic_v<double>,
37        "std::is_arithmetic_v<std::string>: ",
38        std::is_arithmetic_v<std::string>);
39 }

```

```

CHECK WITH TYPE TRAITS 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 WITH TYPE TRAITS 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 WITH TYPE TRAITS 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 in this example has a `static constexpr bool` member named `value`. An expression like line 12's

```
std::is_integral<int>::value
```

evaluates to `true` at compile-time if the type in angle brackets (`int`) is an integral type; otherwise, it evaluates to `false`.

The notation `::value` is commonly used with type-trait classes to access their `true` or `false` values, so C++ has convenient shorthands for using type trait values. These are defined as **variable templates** with names ending in `_v`.⁴¹ Just as function templates specify groups of related functions and class templates specify groups of related classes, variable templates specify groups of related variables. You instantiate a variable template to use it. The variable template corresponding to

```
std::is_integral<int>::value
```

is defined in the C++ standard as⁴²

```
template<class T>
inline constexpr bool is_integral_v = is_integral<T>::value;
```

So, the expression `std::is_integral_v<int>` is equivalent to

```
std::is_integral<int>::value
```

Lines 12, 22 and 34 test type traits and display their `value` members to show the results. The program's other tests use the more convenient `_v` variable templates:

- Lines 13, 14, 15 and 17 use `std::is_integral_v` to check whether various types are **integer types**.
- Lines 24, 26, 28 and 30 use `std::is_floating_point_v` to check whether various types are **floating-point types**.
- Lines 35, 36 and 38 use `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

https://en.cppreference.com/w/cpp/header/type_traits

This table lists the type traits in the same categories and order as [cppreference.com](https://en.cppreference.com).

Type traits by category

Helper classes

<code>bool_constant</code> ¹⁷	<code>false_type</code>	<code>true_type</code>
<code>integral_constant</code>		

41. C++ Core Guidelines, “T.142: Use Template Variables to Simplify Notation.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-var>.

42. C++ Standard, “20.15.3 Header `<type_traits>` Synopsis.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/meta#type.synop>.

Type traits by category (Cont.)

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_signed</code>
<code>is_const</code>	<code>is_empty</code>	<code>is_unsigned</code>
<code>is_volatile</code>	<code>is_polymorphic</code>	<code>is_bounded_array</code> ²⁰
<code>is_trivial</code>	<code>is_abstract</code>	<code>is_unbounded_array</code> ²⁰
<code>is_trivially_copyable</code>	<code>is_final</code> ¹⁴	<code>is_scoped_enum</code> ²³
	<code>is_aggregate</code> ¹⁷	

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_trivially_destructible</code>	<code>is_trivially_destructible</code>
<code>is_trivially_default_constructible</code>	<code>is_assignable</code>	<code>is_nothrow_destructible</code>
<code>is_nothrow_default_constructible</code>	<code>is_trivially_assignable</code>	<code>has_virtual_destructor</code>
<code>is_copy_constructible</code>	<code>is_nothrow_assignable</code>	<code>is_swappable_with</code> ¹⁷
<code>is_trivially_copy_constructible</code>	<code>is_copy_assignable</code>	<code>is_swappable</code> ¹⁷
<code>is_nothrow_copy_constructible</code>	<code>is_trivially_copy_assignable</code>	<code>is_nothrow_swappable_with</code> ¹⁷
	<code>is_nothrow_copy_assignable</code>	<code>is_nothrow_swappable</code> ¹⁷
	<code>is_move_assignable</code>	

Property queries

<code>alignment_of</code>	<code>rank</code>	<code>extent</code>
---------------------------	-------------------	---------------------

Type relationships

<code>is_same</code>	<code>is_layout_compatible</code> ²⁰	<code>is_invocable_r</code> ¹⁷
<code>is_base_of</code>	<code>is_pointer_interconvertible</code> ²⁰	<code>is_nothrow_invocable</code> ¹⁷
<code>is_convertible</code>	<code>base_of</code>	<code>is_nothrow_invocable_r</code> ¹⁷
<code>is_nothrow_convertible</code> ²⁰	<code>is_invocable</code> ¹⁷	

Const-volatility specifiers

<code>remove_cv</code>	<code>remove_volatile</code>	<code>add_const</code>
<code>remove_const</code>	<code>add_cv</code>	<code>add_volatile</code>

References

<code>remove_reference</code>	<code>add_lvalue_reference</code>	<code>add_rvalue_reference</code>
-------------------------------	-----------------------------------	-----------------------------------

Type traits by category (Cont.)

Pointers

<code>remove_pointer</code>	<code>add_pointer</code>
-----------------------------	--------------------------

Sign modifiers

<code>make_signed</code>	<code>make_unsigned</code>
--------------------------	----------------------------

Arrays

<code>remove_extent</code>	<code>remove_all_extents</code>
----------------------------	---------------------------------

Miscellaneous transformations

<code>aligned_storage</code>	<code>conditional</code>	<code>underlying_type</code>
<code>aligned_union</code>	<code>common_type</code>	<code>invoke_result</code> ¹⁷
<code>decay</code>	<code>common_reference</code> ²⁰	<code>void_t</code> ¹⁷
<code>remove_cvref</code> ²⁰	<code>basic_common_reference</code> ²⁰	<code>type_identity</code> ²⁰
<code>enable_if</code>		

Operations on traits

<code>conjunction</code> ¹⁷	<code>disjunction</code> ¹⁷	<code>negation</code> ¹⁷
--	--	-------------------------------------

Functions

Member relationships

<code>is_pointer_interconvertible_with_class</code> ²⁰	<code>is_corresponding_member</code> 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 and 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`.⁴³ 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:

```
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

A concept begins with a `template` header followed by

- the C++20 keyword `concept`,
- the `concept name` (`Numeric`),
- an `equal sign` (`=`) and
- the `constraint expression` (`std::integral<T> || std::floating_point<T>`).

43. Inbal Levi, "Exploration of C++20 Metaprogramming," September 29, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=XgrjybKaIV8>.

The **constraint expression** is a compile-time logical expression that determines whether a template argument satisfies a particular set of requirements—in this case, an integer or floating-point type. Constraint expressions commonly include **predefined concepts**, **type traits** and, as you'll see in Section 15.6.5, **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.⁴⁴ 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:

```
template<typename T, typename U>
    requires std::same_as<T, U>
// rest of template definition
```

In Section 15.12, we'll 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



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. Here's our **multiply** function template using the **custom concept Numeric<T>**:

```
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:

```
template<typename T>
T multiply(T first, T second) requires Numeric<T> {
    return first * second;
}
```

A **trailing requires clause** must be used in two scenarios:⁴⁵

- 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 before the compiler evaluates the **requires** clause.

44. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces,” January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.

45. “What Is the Difference Between the Three Ways of Applying Constraints to a Template?” Accessed April 18, 2023. <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.]

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.⁴⁶ Doing so simplifies the template definition by eliminating the `requires` clause:

```
template<Numeric T>
T multiply(T first, T second) {return first * second;}
```

Now, each parameter must satisfy the concept `Numeric`'s requirements. The function template still has only one type parameter, so the function's arguments must have the same



type; otherwise, a compilation error occurs.



15.6.3 Using Concepts in Abbreviated Function Templates



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.⁴⁷ This includes

- abbreviated function template parameter lists,
- `auto` specified as a function's return type,
- `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.

```
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<concept Numeric 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
```

Fig. 15.7 | Constrained `multiply` abbreviated function template. (Part 1 of 2.)

46. C++ Core Guidelines, “T.13: Prefer the Shorthand Notation for Simple, Single-Type Argument Concepts.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-shorthand>.

47. “Placeholder Type Specifiers.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/auto>.

```

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
```

Fig. 15.7 | Constrained `multiply` abbreviated function template. (Part 2 of 2.)

The key difference between this **abbreviated function template** and the constrained `multiply` function templates in Section 15.4.2 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**:

```

template<Numeric T1, Numeric T2>
auto multiply(T1 first, T2 second) {return first * second;}
```

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

Function templates and overloading are intimately related. When overloaded functions perform syntactically 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 instantiate the template appropriately for each function call. The instantiations all have the same function name, so the compiler uses **overload resolution** to invoke the proper one.

Matching Process for Overloaded Functions

When determining which 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.^{48,49}



48. The compiler's process for resolving function calls is complex. The complete details are discussed in the C++ Standard, “12.2 Overload Resolution.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/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

With C++20, you can use **concepts** in function templates to select overloads based on type requirements.⁵⁰ Consider the standard functions `std::distance` and `std::advance` from the `<iterator>` header—each operates on a container via iterators:

- `std::distance` calculates the number of elements between two iterators.⁵¹
- `std::advance` advances an iterator n positions from its current location.⁵²

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 loop iteration.

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` 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 between the overloads based on the concepts we use to constrain each template's parameters.

```

1 // fig15_08.cpp
2 // Using concepts to select overloads.
3 #include <array>
4 #include <iostream>
5 #include <iterator>
6 #include <list>

```

Fig. 15.8 | Using concepts to select overloads. (Part I of 2.)

-
49. Jeff Preshing, “How C++ Resolves a Function Call.” March 15, 2021. Accessed May 7, 2023. <https://preshing.com/20210315/how-cpp-resolves-a-function-call/>.
 50. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—6 Concept Overloading,” January 31, 2017. Accessed April 18, 2023. <http://wg21.link/p0557r0>.
 51. “`std::distance`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/iterator/distance>.
 52. “`std::advance`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/iterator/advance>.

```

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 template <std::input_iterator Iterator>
12 auto customDistance(Iterator begin, Iterator end) {
13     std::cout << "Called customDistance with input 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 template <std::random_access_iterator Iterator>
27 auto customDistance(Iterator begin, Iterator end) {
28     std::cout << "Called customDistance with random-access iterators\n";
29     return end - begin; // returns a std::ptrdiff_t value
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 input iterators
ints2 number of elements: 3

```

Fig. 15.8 | Using concepts to select overloads. (Part 2 of 2.)

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.

C++ uses the type `std::ptrdiff_t` (line 14) to represent the difference between two pointers or two iterators. The `std::distance` algorithm returns this type, so we do as well in our `customDistance` implementations.

Lines 33–34 define an array and a `list`. Line 36 calls `customDistance` for the array `ints1`, which has **random-access iterators**. Our function that requires input iterators could receive random-access iterators. However, when multiple function templates satisfy a function call, the compiler calls the most constrained version.^{53,54} So line 36 calls the $O(1)$ version of `customDistance` (lines 26–30). Based on their definitions, the compiler knows that `random_access_iterator` is more constrained than `input_iterator`. 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.

Concepts C 15.6.5 requires Expressions

CG The C++ Core Guidelines recommend using standard concepts.⁵⁵ For custom requirements that cannot be expressed via standard concepts in a `requires` clause, you can use a **requires expression**, which has two forms:

```
requires {
    requirement-definitions
}
```

or

```
requires (parameter-list) {
    requirements-definitions that optionally use the parameters
}
```

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

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`:⁵⁶

```
1 template<class T>
2 concept range =
3     requires(T& t) {
4         std::ranges::begin(t);
5         std::ranges::end(t);
6     };
```

- 53. Hendrik Niemeyer, “An Introduction to C++20’s Concepts,” July 25, 2020. Accessed April 18, 2023. https://www.youtube.com/watch?v=N_kPd20K1L8.
- 54. C++ Standard, “13.5 Template Constraints.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/temp.constr>.
- 55. C++ Core Guidelines, “T.11: Whenever Possible Use Standard Concepts.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts>.
- 56. C++ Standard, “24.4.2 Ranges.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/ranges#range.range>.

The requirements can reference the expression's parameter(s) and template parameter(s) from the `template` header. This **range concept** defines a `T`& parameter `t`. Lines 4–5 define **simple requirements** indicating that `t` is a range only if we can get `t`'s `begin` iterator and `end sentinel` by passing `t` to functions `std::ranges::begin` and `std::ranges::end`, respectively. If either of these **simple requirements** does not compile, 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:

```
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 modulus (%) 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. The `type trait is_arithmetic` tests for built-in arithmetic types—for custom-class types, it always evaluates to `false`.

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 type argument have a nested type called `value_type` (which is the case for standard library containers), you might define a concept with a type requirement like

```
template<typename T>
concept HasValueType = requires {
    typename T::value_type;
};
```

A type `T` would satisfy `HasValueType` only if `T::value_type` is a valid type. If type `T` does not contain a nested `value_type`, this requirement would evaluate to `false`.

Compound Requirements

A **compound requirement** allows you to specify an expression that also has requirements on its result. Such requirements have the form

```
{ expression } -> return-type-requirement
```

For example, the standard library `<iterator>` header's **incrementable** concept contains the following `requires` expression, specifying that an `incrementable` iterator must support the postincrement operator `(++)`:⁵⁷

57. C++ Standard, “23.3.4.5 Concept `incrementable`.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/iterator.concepts#iterator.concept.inc>.

```
requires(I i) {
    { i++ } -> same_as<I>;
}
```

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 `i`'s type. The right brace in a compound requirement optionally may be followed by `noexcept` to indicate that the expression must be `noexcept`. When we introduced `std::same_as` earlier, we used two template arguments, but we used only one here. When you specify a constraint on an expression's result, the compiler inserts the expression's type into the constraint as the first type parameter. So there actually are two type parameters in the preceding constraint—in this case, both are type `I`.⁵⁸

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 preincrementing, and the expression's result must have the same type as a reference to the type parameter `I`.

Nested Requirement

A `nested requirement` is a `requires` clause nested in a `requires` expression. You'd use nested requirements 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::ranges::range` concept's `requires` expression and placed it directly in a `requires` clause:

```
template<typename T>
requires requires(const 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.
- The second `requires` introduces the `requires expression`.

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.**

58. C++ Standard, “13.2 Template Parameters.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/temp.param#4>.



15.6.6 C++20 Exposition-Only Concepts

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**.⁵⁹ They often 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 `->`:⁶⁰

```
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:⁶¹

```
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*.

The standard uses the **31 exposition-only concepts** shown in the following table. You can find each through the standard’s “Index of library concepts.”⁶²

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>	

59. “Exposition-Only in the C++ Standard?” Answered December 28, 2015. Accessed April 18, 2023. <https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard>.

60. C++ Standard, “24.5.1 Helper Concepts.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/range.utility>.

61. C++ Standard, “24.6.4.2 Class Template *iota_view*.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/range.factories#range.iota.view-2>.

62. C++ Standard, “Index of Library Concepts.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/conceptindex>.

31 exposition-only concepts (Cont.)

Comparisons in the language support library

compares-as *partially-ordered-with*

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 C++ standard, 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** (Section 15.13.3 shows **constexpr**). For an overview of the progression through these technologies, see the blog post, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts."⁶³

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 **instantiating function templates** based on a function call's argument types.

Before concepts, developers had to be significantly more familiar with template specialization rules. Since template debugging is nontrivial, this made the code more error-prone. 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 invalid code. It would then ignore that code, removing it from the overload set. **SFINAE (substitution failure is not an error)**^{64,65,66} describes how the compiler discards invalid template code as it determines the correct function to call. SFINAE prevents the compiler from immediately generating potentially lengthy lists of errors when first instantiating a template. The compiler generates error messages only if it cannot find a match for the function call in the overload set.

Tag Dispatch

Using the **tag-dispatch**⁶⁷ technique, you can tell the compiler the version of an overloaded function to call based on template type parameters and properties of those types. Bjarne Stroustrup—in his paper “Concepts: The Future of Generic Programming”—refers to

-
- 63. Bartłomiej Filipek, “Notes on C++ SFINAE, Modern C++ and C++20 Concepts,” April 20, 2020. Accessed April 18, 2023. <https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html>.
 - 64. “SubstitutionFailure Is Not an Error.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.
 - 65. Filipek, “Notes on C++ SFINAE, Modern C++ and C++20 Concepts.”
 - 66. David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.
 - 67. “Generic Programming—Tag Dispatching.” Accessed April 18, 2023. https://www.boost.org/community/generic_programming.html#tag_dispatching.

properties of types as “concepts” and discusses how C++20 concept-based overloading (Section 15.6.4) can replace tag dispatch.⁶⁸

15.7 Testing C++20 Concepts with static_assert

Concepts produce compile-time `bool` values, which you can test at compile-time with a `static_assert` declaration.⁶⁹ `static_assert` was developed to add compile-time assertion support for reporting incorrect usage of template libraries.⁷⁰ Figure 15.9 tests our custom `Numeric` concept from Section 15.6.1 in a `multiply` function template that is not concept constrained.⁷¹



```

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 template<typename T>
10 auto multiply(T a, T b) {
11     static_assert(Numeric<T>);
12     return a * b;
13 }
14
15 int main() {
16     using namespace std::string_literals;
17     multiply(2, 5); // OK: int is Numeric
18     multiply(2.5, 5.5); // OK: double is Numeric
19     multiply("2"s, "5"s); // error: string is not Numeric
20 }
```

```

fig15_09.cpp:11:4: error: static_assert failed
    static_assert(Numeric<T>);
    ^
          ~~~~~~
fig15_09.cpp:19:4: note: in instantiation of function template specialization
'multiply<std::basic_string<char>>' requested here
    multiply("2"s, "5"s); // error: string is not Numeric
    ^
```

Fig. 15.9 | Testing custom concepts with `static_assert`. (Part 1 of 2.)

- 68. Bjarne Stroustrup, “Concepts: The Future of Generic Programming (Section 6).” Accessed April 18, 2023. https://www.stroustrup.com/good_concepts.pdf.
- 69. C++ Core Guidelines, “T.150: Check That a Class Matches a Concept Using `static_assert`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-check-class>.
- 70. Robert Klarer, John Maddock, Beman Dawes and Howard Hinnant, “Proposal to Add Static Assertions to the Core Language (Revision 3),” October 20, 2004. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html>.
- 71. With concepts, you do not need to use `static_assert`, as shown here.

```

fig15_09.cpp:11:18: note: because 'std::basic_string<char>' does not satisfy
'Numeric'
    static_assert(Numeric<T>);
               ^
fig15_09.cpp:7:24: note: because 'std::basic_string<char>' does not satisfy
'integral'
concept Numeric = std::integral<T> || std::floating_point<T>;
               ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:102:24: note: because 'is_integral_v<std::basic_string<char> >' 
evaluated to false
    concept integral = is_integral_v<Tp>;
               ^
fig15_09.cpp:7:44: note: and 'std::basic_string<char>' does not satisfy
'floating_point'
concept Numeric = std::integral<T> || std::floating_point<T>;
               ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:111:30: note: because 'is_floating_point_v<std::basic_string<char> >' 
evaluated to false
    concept floating_point = is_floating_point_v<Tp>;
               ^
fig15_09.cpp:12:13: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
    return a * b;
           ~ ^ ~
2 errors generated.

```

Fig. 15.9 | Testing custom concepts with `static_assert`. (Part 2 of 2.)



When the `static_assert` argument is `false`, the compiler outputs an error message that tells you what and where the error is. If the `static_assert` argument is `true`, the compiler does not output any messages—it simply continues compiling the code. The expression in line 11

```
static_assert(Numeric<T>);
```

checks whether `multiply`'s argument type (`T`) satisfies the `Numeric` concept's requirements. If it does, `Numeric<T>` evaluates to `true`, and the compiler continues compiling the code. When we call `multiply` from lines 17 and 18, `Numeric<T>` evaluates to `true` because the argument types `int` and `double` both satisfy the concept `Numeric`.

However, when we call `multiply` in line 19 with string-object literals, `Numeric<T>` in line 11 evaluates to `false` because a `string` is not `Numeric`. The output window shows the error messages produced by Clang C++. We highlighted the key messages in bold and added blank lines for readability.

For the `Numeric` concept, the compiler tells you

```
note: because 'std::basic_string<char>' does not satisfy 'Numeric'
```

The messages also provide more detail, saying that

```
note: because 'std::basic_string<char>' does not satisfy 'integral'
```

and that

```
note: and 'std::basic_string<char>' does not satisfy 'floating_point'
```

A **static_assert** declaration optionally may specify as a second argument a string to include in the error message for a **false** assertion.

15.8 Creating a Custom Algorithm

We saw in Chapters 13 and 14 that the standard 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 the custom **NumericInputRange** concept in lines 14–17, which we describe after the figure.

```
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 <list>
10 #include <ranges>
11 #include <vector>
12
13 // concept for an input range containing integer or floating-point values
14 template<typename T>
15 concept NumericInputRange = std::ranges::input_range<T> &&
16     (std::integral<typename T::value_type> || 
17      std::floating_point<typename T::value_type>);
18
19 // calculate the average of a NumericInputRange's elements
20 auto average(NumericInputRange auto const& range) {
21     long double total{0};
22
23     for (auto i{range.begin()}; i != range.end(); ++i) {
24         total += *i; // dereference iterator and add value to total
25     }
26
27     // divide total by the number of elements in range
28     return total / std::ranges::distance(range);
29 }
30
```

Fig. 15.10 | A custom algorithm to calculate the average of a numeric input range. (Part I of 2.)

```

31 int main() {
32     std::ostream_iterator<int> outputInt(std::cout, " ");
33     const std::array ints{1, 2, 3, 4, 5};
34     std::cout << "array ints: ";
35     std::ranges::copy(ints, outputInt);
36     std::cout << "\naverage of ints: " << average(ints);
37
38     std::ostream_iterator<double> outputDouble(std::cout, " ");
39     const std::vector doubles{10.1, 20.2, 35.3};
40     std::cout << "\nvector doubles: ";
41     std::ranges::copy(doubles, outputDouble);
42     std::cout << "\naverage of doubles: " << average(doubles);
43
44     std::ostream_iterator<long double> outputLongDouble(std::cout, " ");
45     const std::list longDoubles{10.1L, 20.2L, 35.3L};
46     std::cout << "\nlist longDoubles: ";
47     std::ranges::copy(longDoubles, outputLongDouble);
48     std::cout << "\naverage of longDoubles: " << average(longDoubles)
49     << "\n";
50 }

```

```

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

C Custom NumericInputRange Concept

The **custom NumericInputRange** concept (lines 14–17) checks

- whether a type satisfies the `input_range` concept (line 15), so the argument supports at least `input iterators` for reading its elements, and
- whether a range's elements satisfy the `std::integral` or `std::floating_point` concepts (lines 16–17), so they can be used in calculations.

Recall from Section 13.4.1 that each standard container has a nested `value_type`, which indicates the container's element type. We use this to check whether the element type satisfies this concept's requirements. In lines 16 and 17, the notation

`typename T::value_type`

indicates that `T::value_type` is an alias for the range's element type.

Custom average Algorithm

Lines 20–29 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 21) to store the sum of the elements.

Lines 23–25 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 28 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 33, 39 and 45 define an array of ints, a vector of doubles and a list of long doubles, respectively. Lines 36, 42 and 48 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 using templates to implement any class representing a container of values or range of values.⁷² So, here, we'll define `MyArray` as a class template and enhance it using various standard library conventions.⁷³ We'll also define **custom iterators** to use `MyArray` objects with various standard library algorithms.



Our goal here is to give you a sense of what's involved in creating standard-library-like containers. Achieving full standard library compatibility and backward compatibility with prior C++ language versions involves many **conditional compilation directives** beyond this book's scope. If you'd like to build reusable standard-library-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 to instantiate it. Also, defining member functions inside the class template definition simplifies the syntax, as you do not need template headers for 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 discussed in Section 13.5, pointers into such arrays satisfy all the requirements of random-access iterators. However, for this example, 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,⁷⁴ which defines two iterator classes:

72. C++ Core Guidelines, "T.3: Use Templates to Express Containers and Ranges." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-cont>.

73. Jonathan Boccaro, "Make Your Containers Follow the Conventions of the STL," April 24, 2018. Accessed April 18, 2023. <https://www.fluentcpp.com/2018/04/24/following-conventions-stl/>.

74. "array Standard Header." Latest commit (i.e., when the file was last updated) February 24, 2021. Accessed April 18, 2023. <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 Microsoft's implementation at

<https://github.com/microsoft/STL/blob/main/stl/inc/array>

We defined the following custom iterator classes:

- Our **ConstIterator** class represents a **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

<https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array>

and

<https://github.com/llvm/llvm-project/blob/main/libcxx/include/array>

Why We Implemented Bidirectional Rather Than Random-Access Iterators

Recall from our introduction to iterators in Section 13.5 that various levels of iterators are supported by standard library containers. The most powerful are **contiguous iterators**, which are **random-access iterators** for containers that guarantee contiguous memory. Most standard library algorithms operate on ranges of container elements. Only 12 require **random-access iterators**—shuffle and 11 sorting-related algorithms—and none require **contiguous iterators**. **MyArray**'s **bidirectional** iterators enable most standard library algorithms to process **MyArray** objects. Random-access iterators have many additional requirements⁷⁵ shown in the table below. As an exercise, you could enhance **MyArray**'s custom iterators with these capabilities to make them **random-access iterators**.

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</code> or <code>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> .

(continued...)

75. C++ Standard, “23.3.4.13 Concept `random_access_iterator`.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/iterator.concept.random.access>.

Random-access iterator operation	Description
$p \leq p1$	Return <code>true</code> if iterator p is less than or equal to iterator $p1$ (that is, iterator p is before or at the same location as iterator $p1$ in the container); otherwise, return <code>false</code> .
$p > p1$	Return <code>true</code> if iterator p is greater than iterator $p1$ (that is, iterator p is after iterator $p1$ in the container); otherwise, return <code>false</code> .
$p \geq p1$	Return <code>true</code> if iterator p is greater than or equal to iterator $p1$ (that is, iterator p is after or at the same location as iterator $p1$ in the container); otherwise, return <code>false</code> .

Basic Iterator Requirements

All iterators must support:^{76,77,78}

- 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 15–77 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 15), representing a **MyArray**’s element type. Each **ConstIterator** points to an element of that type.

```

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
6 #include <algorithm>
7 #include <compare>
```

Fig. 15.11 | Class template **MyArray** with custom iterators implemented by class templates **ConstIterator** and **Iterator**—class **ConstIterator**. (Part 1 of 3.)

-
76. “C++ Named Requirements: LegacyIterator.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/named_req/Iterator.
 77. Triangles, “Writing a Custom Iterator in Modern C++,” December 19, 2020. Accessed April 18, 2023. <https://internalpointers.com/post/writing-custom-iterators-modern-cpp>.
 78. David Gorski, “Custom STL Compatible Iterators,” March 2, 2019. Accessed April 18, 2023. <https://davidgorski.ca/posts/stl-iterators/>.

```

8 #include <initializer_list>
9 #include <iostream>
10 #include <iterator>
11 #include <stdexcept>
12 #include <utility>
13
14 // class template ConstIterator for a MyArray const iterator
15 template <typename T>
16 class ConstIterator {
17 public:
18     // public iterator nested type names
19     using iterator_category = std::bidirectional_iterator_tag;
20     using difference_type = std::ptrdiff_t;
21     using value_type = T;
22     using pointer = const value_type*;
23     using reference = const value_type&;
24
25     // default constructor
26     ConstIterator() = default;
27
28     // initialize a ConstIterator with a pointer into a MyArray
29     ConstIterator(pointer p) : m_ptr{p} {}
30
31     // OPERATIONS ALL ITERATORS MUST PROVIDE
32     // increment the iterator to the next element and
33     // return a reference to the iterator
34     ConstIterator& operator++() noexcept {
35         ++m_ptr;
36         return *this;
37     }
38
39     // increment the iterator to the next element and
40     // return the iterator before the increment
41     ConstIterator operator++(int) noexcept {
42         ConstIterator temp{*this};
43         ++(*this);
44         return temp;
45     }
46
47     // OPERATIONS INPUT ITERATORS MUST PROVIDE
48     // return a const reference to the element m_ptr points to
49     reference operator*() const noexcept {return *m_ptr;}
50
51     // return a const pointer to the element m_ptr points to
52     pointer operator->() const noexcept {return m_ptr;}
53
54     // <=> operator automatically supports equality/relational operators.
55     // Only == and != are needed for bidirectional iterators.
56     // This implementation would support the <, <=, > and >= required
57     // by random-access iterators.
58     auto operator<=>(const ConstIterator& other) const = default;

```

Fig. 15.11 | Class template MyArray with custom iterators implemented by class templates ConstIterator and Iterator—class ConstIterator. (Part 2 of 3.)

```

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
68      // decrement the iterator to the previous element and
69      // return the iterator before the decrement
70      ConstIterator operator--(int) noexcept {
71          ConstIterator temp{*this};
72          --(*this);
73          return temp;
74      }
75  private:
76      pointer m_ptr{nullptr};
77  };
78

```

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 19–23 define type aliases for the **nested type names** that the C++ standard library expects in iterator classes:⁷⁹

- **iterator_category**: The iterator’s category (Section 13.5.2), 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 `ConstIterator`’s data member (line 76) 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 **defaulted constructor** (line 26) that initializes a `ConstIterator`’s `m_ptr` member using its in-class initializer (`nullptr`; line 76), and a constructor that initializes a `ConstIterator` from a pointer to an element (line 29). The class’s **copy and move constructors** are autogenerated.

79. C++ Standard, “23.3.2.3 Iterator Traits.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/iterators#iterator.traits>.

`++ Operators`

Lines 34–37 and 41–45 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 49 and 52). 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 58) 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 63–66 and 70–74 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 81–137 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 81), representing a `MyArray`’s element type. Class `Iterator` inherits from class `ConstIterator<T>` (line 82).

```

79 // class template Iterator for a MyArray non-const iterator;
80 // redefines several inherited operators to return non-const results
81 template <typename T>
82 class Iterator : public ConstIterator<T> {
83 public:
84     // public iterator nested type names
85     using iterator_category = std::bidirectional_iterator_tag;
86     using difference_type = std::ptrdiff_t;
87     using value_type = T;
88     using pointer = value_type*;
89     using reference = value_type&;
90
91     // inherit ConstIterator constructors
92     using ConstIterator<T>::ConstIterator;
93

```

Fig. 15.11 | Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `Iterator`. (Part I of 2.).

```
94 // OPERATIONS ALL ITERATORS MUST PROVIDE
95 // increment the iterator to the next element and
96 // return a reference to the iterator
97 Iterator& operator++() noexcept {
98     ConstIterator<T>::operator++(); // call base-class version
99     return *this;
100 }
101
102 // increment the iterator to the next element and
103 // return the iterator before the increment
104 Iterator operator++(int) noexcept {
105     Iterator temp{*this};
106     ConstIterator<T>::operator++(); // call base-class version
107     return temp;
108 }
109
110 // OPERATIONS INPUT ITERATORS MUST PROVIDE
111 // return a reference to the element m_ptr points to; this
112 // operator returns a non-const reference for output iterator support
113 reference operator*() const noexcept {
114     return const_cast<reference>(ConstIterator<T>::operator*());
115 }
116
117 // return a pointer to the element m_ptr points to
118 pointer operator->() const noexcept {
119     return const_cast<pointer>(ConstIterator<T>::operator->());
120 }
121
122 // OPERATIONS BIDIRECTIONAL ITERATORS MUST PROVIDE
123 // decrement the iterator to the previous element and
124 // return a reference to the iterator
125 Iterator& operator--() noexcept {
126     ConstIterator<T>::operator--(); // call base-class version
127     return *this;
128 }
129
130 // decrement the iterator to the previous element and
131 // return the iterator before the decrement
132 Iterator operator--(int) noexcept {
133     Iterator temp{*this};
134     ConstIterator<T>::operator--(); // call base-class version
135     return temp;
136 }
137 };
138 
```

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 85–89 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 88–89 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 92 simply inherits the base class's constructors.

++ and -- Operators

Lines 97–100, 104–108, 125–128 and 132–136 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 113–115 and 118–120 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 114 and 119 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. In general, `const_cast` should be used only if you know the original data is not constant⁸⁰—as in class `Iterator`.

CG

15.9.3 Class Template MyArray

Lines 141–211 of Fig. 15.11 define our simplified `MyArray` class template. We removed some overloaded operators and special member functions presented in Chapter 11 to focus on the `container` and its `iterators`. To mimic `std::array`, we define `MyArray` as an `aggregate type` (Section 9.21), which requires all non-`static` data to be `public`—a `struct` has `public` members by default. We also store the data in a fixed-size built-in array (line 210) allocated at compile-time, rather than using dynamic memory.

```

139 // class template MyArray contains a fixed-size T[SIZE] array;
140 // MyArray is an aggregate type with public data, like std::array
141 template <typename T, size_t SIZE>
142 struct MyArray {
143     // type names used in standard library containers
144     using value_type = T;
145     using size_type = size_t;
146     using difference_type = ptrdiff_t;
147     using pointer = value_type*;
148     using const_pointer = const value_type*;
149     using reference = value_type&;
150     using const_reference = const value_type&;
151
152     // iterator type names used in standard library containers
153     using iterator = Iterator<T>;
154     using const_iterator = ConstIterator<T>;
155     using reverse_iterator = std::reverse_iterator<iterator>;
156     using const_reverse_iterator = std::reverse_iterator<const_iterator>;

```

Fig. 15.11 | Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `MyArray`. (Part 1 of 3.)

80. C++ Core Guidelines, “ES.50: Don’t cast away `const`.” Accessed March 10, 2022. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>.

```
157 // Rule of Zero: MyArray's special member functions are autogenerated
158
159 constexpr size_type size() const noexcept {return SIZE;} // return size
160
161
162 // member functions that return iterators
163 iterator begin() {return iterator{&m_data[0]};}
164 iterator end() {return iterator{&m_data[0] + size();}}
165 const_iterator begin() const {return const_iterator{&m_data[0]};}
166 const_iterator end() const {
167     return const_iterator{&m_data[0] + size();}
168 }
169 const_iterator cbegin() const {return begin();}
170 const_iterator cend() const {return end();}
171
172 // member functions that return reverse iterators
173 reverse_iterator rbegin() {return reverse_iterator{end();}}
174 reverse_iterator rend() {return reverse_iterator{begin();}}
175 const_reverse_iterator rbegin() const {
176     return const_reverse_iterator{end();}
177 }
178 const_reverse_iterator rend() const {
179     return const_reverse_iterator{begin();}
180 }
181 const_reverse_iterator crbegin() const {return rbegin();}
182 const_reverse_iterator crend() const {return rend();}
183
184 // autogenerated three-way comparison operator
185 auto operator<=>(const MyArray& t) const noexcept = default;
186
187 // overloaded subscript operator for non-const MyArrays;
188 // reference return creates a modifiable lvalue
189 T& operator[](size_type index) {
190     // check for index out-of-range error
191     if (index >= size()) {
192         throw std::out_of_range{"Index out of range"};
193     }
194
195     return m_data[index]; // reference return
196 }
197
198 // overloaded subscript operator for const MyArrays;
199 // const reference return creates a non-modifiable lvalue
200 const T& operator[](size_type index) const {
201     // check for subscript out-of-range error
202     if (index >= size()) {
203         throw std::out_of_range{"Index out of range"};
204     }
205
206     return m_data[index]; // returns copy of this element
207 }
208 }
```

Fig. 15.11 | Class template MyArray with custom iterators implemented by class templates ConstIterator and Iterator—class MyArray. (Part 2 of 3.)

```

209     // Like std::array the data is public to make this an aggregate type
210     T m_data[SIZE]; // built-in array of type T with SIZE elements
211 };
212

```

Fig. 15.11 | Class template MyArray with custom iterators implemented by class templates ConstIterator and Iterator—class MyArray. (Part 3 of 3.)

MyArray's template Header

The template header (line 141) 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 210 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 144–156 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 at least **bidirectional iterators**:⁸¹

- **value_type**: The container's element type (T).
- **size_type**: The type 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 adaptor std::reverse_iterator** creates a reverse-iterator type from its iterator type argument, which must be at least bidirectional.
- **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 iterator type argument, which must be at least bidirectional.

81. C++ Standard, “Table 73: Container Requirements.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/container.requirements#tab:container.req>.

MyArray Member Functions That Return Iterators

Lines 163–182 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 163–168:

- `begin` (line 163) returns an iterator pointing to the MyArray’s first element.
- `end` (line 164) returns an iterator pointing to one past the MyArray’s last element.
- `begin` (line 165) is a `const`-qualified overload that returns a `const_iterator` pointing to the MyArray’s first element.
- `end` (line 166–168) is a `const`-qualified overload that returns a `const_iterator` pointing to one past the MyArray’s last element.

The other member functions that return iterators call these `begin` and `end` functions:

- `cbegin` and `cend` (lines 169–170) call the versions of `begin` and `end` that return `const` iterators.
- `rbegin` and `rend` (lines 173–174) produce `reverse_iterators` based on the iterators returned by the `non-const` versions of `begin` and `end`.
- `rbegin` and `rend` (lines 175–180) 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 181–182) return the `const_reverse_iterators` from calling the `const` versions of `rbegin` and `rend`.

MyArray Overloaded Operators

Lines 185, 189–196 and 200–207 define MyArray’s overloaded operators. Though we won’t use it in this example, the **compiler-generated three-way comparison operator** (line 185) enables you to compare entire MyArray objects of the same element type and size. The overloaded `operator[]` member functions are identical to those in Chapter 11’s MyArray class, but we now call the `size` member function (lines 191 and 202) 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:

```
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. MyArray does not define a constructor that can receive any number of arguments. However, we can define a **deduction guide**⁸² (lines 214–215) that shows the compiler how to deduce a MyArray’s type from a braced initializer. Then the compiler can use **aggregate initialization** (e.g., line 15 in Fig. 15.12) to place the initializers into our MyArray aggregate type’s built-in array data member.

82. “Class Template Argument Deduction.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```

213 // deduction guide to enable MyArrays to be brace initialized
214 template<typename T, std::same_as<T>... Us>
215 MyArray(T first, Us... rest) -> MyArray<T, 1 + sizeof...(Us)>;

```

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 15.12. Line 214 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 of the same type as `T`, specified by `std::same_as<T>... Us`. The `...` indicates a **parameter pack**. A **parameter pack**'s name (`Us`) is typically plural because it can represent multiple items.

To the left of the `->` in line 215, 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 `Us...` 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...(Us)`

In this expression,

- `1` is the initializer list's minimum number of initializers and
- `sizeof...(Us)` uses the compile-time **sizeof... operator** to determine the additional number of initializers the compiler placed in the parameter pack `Us`.

Our deduction guide is based on those provided by GNU and Clang for their `std::array` implementations. You can view GNU's and Clang's deduction guides in their respective `<array>` headers:

`https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array`

and

`https://github.com/llvm/llvm-project/blob/main/libcxx/include/array`

15.9.5 Using `MyArray` 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`.

```
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    using namespace std::string_literals; // for string object literals
24    MyArray strings{"red"s, "orange"s, "yellow"s};
25    std::cout << "\nstrings: ";
26    std::ranges::copy(strings, outputString);
27
28    std::cout << "\n\nDisplaying a MyArray with a range-based for "
29        << "statement, which requires input iterators:\n";
30    for (const auto& item : doubles) {
31        std::cout << item << " ";
32    }
33
34    std::cout << "\n\nCopying a MyArray with std::ranges::copy, "
35        << "which requires an input range and an output iterator:\n";
36    MyArray<std::string, strings.size()> strings2{};
37    std::ranges::copy(strings, strings2.begin());
38    std::cout << "strings2 after copying from strings: ";
39    std::ranges::copy(strings2, outputString);
40
41    std::cout << "\n\nFinding min and max elements in a MyArray "
42        << "with std::ranges::minmax_element, which requires "
43        << "a forward range:\n";
44    auto [min, max] {std::ranges::minmax_element(strings)};
45    std::cout << "min and max elements of strings are: "
46        << *min << ", " << *max;
47
48    std::cout << "\n\nReversing a MyArray with std::ranges::reverse, "
49        << "which requires a bidirectional range:\n";
50    std::ranges::reverse(ints);
```

Fig. 15.12 | Using MyArray with range-based for and with C++ standard library algorithms.
(Part I of 2.)

```

51     std::cout << "ints after reversing elements: ";
52     std::ranges::copy(ints, outputInt);
53     std::cout << "\n";
54 }

```

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 an input range and output iterator:
 strings2 after copying from strings: red orange yellow

Finding min and max elements in a MyArray with `std::ranges::minmax_element`, which requires a forward range:
 min and max elements of strings are: orange, yellow

Reversing a MyArray with `std::ranges::reverse`, which requires a bidirectional range:
 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 2 of 2.)

Creating MyArrays and Displaying Them with `std::ranges::copy`

In lines 13–26, we create three MyArrays (lines 15, 19 and 24), using class template argument deduction to infer their element types and sizes. Lines 17, 21 and 26 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`. MyArrays are compatible with this algorithm because they have more powerful `bidirectional iterators`.

Displaying a MyArray with a Range-Based For Statement

Lines 30–32 display the MyArray doubles using a range-based for, which requires only `input iterators`, so it works with any iterable object, including MyArrays. The doubles MyArray is not a `const` object, so 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 the `std::as_const` function. For example, this example’s range-based for does not modify doubles’ elements, so we could have written line 30 as

```
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 36 creates a new MyArray into which we’ll copy the MyArray strings’ elements. Line 37 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 type with `std::ranges::copy`.

Finding the Minimum and Maximum Elements in a `MyArray` with `std::ranges::minmax_element`

Line 44 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`.

Reversing a `MyArray` with `std::ranges::reverse`

Line 50 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**:

```
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 'MyArray<int, 7> &' does not satisfy 'random_access_range'

and

because 'iterator_t<MyArray<int, 7> &>' (aka 'Iterator<int>') does not satisfy 'random_access_iterator'

```
test.cpp:9:4: error: no matching function for call to object of type 'const
std::ranges::__sort_fn'
    std::ranges::sort(integers);
    ^~~~~~
/usr/bin/.../lib/gcc/x86_64-linux-gnu/10/.../.../.../include/c++/10/bits/rang-
es_algo.h:2032:7: 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:2028:14: note: because 'MyArray<int, 7> &' does not satisfy 'ran-
dom_access_range'
    template<random_access_range _Range,
    ^
(continued...)
```



```

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
range_access.h:934:37: note: because 'iterator_t<MyArray<int, 7> &>' (aka 'It-
erator<int>') does not satisfy 'random_access_iterator'
    = 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:614:10: note: because 'derived_from<__detail::__iter_con-
cept<Iterator<int> >, std::random_access_iterator_tag>' evaluated to false
    && derived_from<__detail::__iter_concept<_Iter>,
                                         ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:67:28: note: because '__is_base_of(std::random_access_iterator_tag,
std::bidirectional_iterator_tag)' evaluated to false
    concept derived_from = __is_base_of(_Base, _Derived)
                                         ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2019:7: note: candidate function template not viable: requires at
least 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 adaptor class template begins with:

```
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 instantiates the **stack** class template for type **int** and uses the resulting specialization to instantiate the object named **values**. The stack's elements are stored in a **deque<int>**.

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. You can use default type arguments for template type parameters in function templates.

15.11 Variable Templates

You've instantiated function templates and class templates to define groups of related functions and classes, respectively. You can use **variable templates** to define groups of related variables. You used several predefined variable templates in Fig. 15.6's type traits demonstration. The C++ standard library has convenient variable templates for accessing each type trait's **value** member. For example, the variable template:

```
is_arithmetic_v<T>
```

is defined in the standard as

```
template<class T>
inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
```

This variable template defines an `inline bool` variable that evaluates to a **compile-time constant** (indicated by `constexpr`). C++ added `inline` variables to better support header-only libraries, which can be included in multiple source-code files (i.e., translation units) within the same application.⁸³ 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 translation units within the same application.⁸⁴



15.12 Variadic Templates and Fold Expressions

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. **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

The `tuple` class (from header `<tuple>`) is a **variadic-class-template generalization of class template pair** (introduced in Section 13.11). A `tuple` is a collection of related values, possibly of mixed types. Figure 15.13 demonstrates several `tuple` capabilities. Line 9's `using` declaration defines the alias `Part` for the type:

```
std::tuple<int, std::string, int, double>
```

This is known as an **alias declaration** and enables you to create a convenient name for a complex type.

```

1 // fig15_13.cpp
2 // Manipulating tuples.
3 #include <format>
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
```

Fig. 15.13 | Manipulating tuples. (Part I of 2.)

83. Alex Pomeranz, “6.8—Global Constants and Inline Variables,” January 3, 2020. Accessed April 18, 2023. <https://www.learncpp.com/cpp-tutorial/global-constants-and-inline-variables/>.

84. “`inline` Specifier.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/inline>.

```

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 << std::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 << std::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 << std::format("A Part tuple has {} elements\n",
44         std::tuple_size_v<Part>); // get the tuple size
45 }

```

```

Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
Part number: 2, Tool: Screwdriver, Quantity: 106, Price: 6.99

Accessing a tuple's elements by index number:
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
A Part tuple has 4 elements

```

Fig. 15.13 | Manipulating tuples. (Part 2 of 2.)

Line 9's `using` declaration creates a **type alias**—a convenient name for a complex type. The alias `Part` represents a tuple for a hardware part's inventory. Each `Part` contains

- 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 and 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.

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

```
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

Using **structured bindings** (Section 14.4.5), you can **unpack a tuple** to access its elements. 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`.

Using `get<type>` to Obtain a tuple Member By Type

You can use `get` with a type argument to access a `tuple` member of a specific type if the `tuple` contains **only one member of that type**. For example, the following statement gets the hammer `tuple`'s `string` member:

```
auto partName{get<std::string>(hammer)};
```

However, the statement

```
auto partNumber{get<int>(hammer)};
```

would generate a compilation error because the call is **ambiguous**—the hammer `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

<https://en.cppreference.com/w/cpp/utility/tuple/tuple>

For other `tuple`-related utilities defined in the `<tuple>` header, see

<https://en.cppreference.com/w/cpp/utility/tuple>

Other <code>tuple</code> class template features	Description
comparisons	Tuples containing the same number of members can be compared to one another using the relational and equality operators, assuming their elements support the appropriate 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.
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 (=) to copy <code>tuple</code> elements in the right operand into a <code>tuple</code> of the same type in the left operand.
move assignment	Uses the assignment operator (=) to move <code>tuple</code> elements in the right operand into a <code>tuple</code> of the same type in the left operand.

15.12.2 Variadic Function Templates and an Intro to 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 **fold expression** to eliminate the recursion.

Section 15.12.7 shows how to test whether all the arguments have the same type.

```

1 // fig15_14.cpp
2 // Variadic function templates.
3 #include <format>
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 }
```

Fig. 15.14 | Variadic function templates. (Part 1 of 2.)

```

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     << std::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     << std::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

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 `typename...` introduces a variadic template's **parameter pack** representing **any number of arguments**. In the function's parameter list (line 15), the variable-length parameter must appear last and is denoted with `...` after its type (`RemainingItems`). Note

that the `...` position is different in the template header and 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 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 `first` in the parameter `first` and `2` in the **parameter pack** `theRest`. The function then adds `1` to the result of calling `sum` with the **parameter-pack expansion** (`...`). The parameter pack contains only `2`, so this call becomes `sum(2)`, invoking `sum`’s one-argument version and 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 `sum(1, 2)` call becomes

```
sum(1, sum(2))
```

or more precisely

```
sum<int, int>(1, sum<int>(2))
```

The compiler has all the values used in the calculation, so it can perform the calculations and inline them in the program as compile-time constants, **eliminating execution-time function-call overhead**.⁸⁵

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, the 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)`.

^{85.} “Template Metaprogramming—Compile-Time Code Optimization.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Template_metaprogramming#Compile-time_code_optimization.

- In the call `sum(2, 3)`, the 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

```
sum("s"s, sum("u"s, sum("m"s)))
```

producing the string "sum".

Fold Expressions

Fold expressions provide a convenient notation for repeatedly applying a binary operator to all the elements in a variadic template's **parameter pack**.^{86,87,88} 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).

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.** A **binary-left-fold expression** has 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)
```

-
86. Jonathan Boccaro, “C++ Fold Expressions 101,” March 12, 2021. Accessed April 18, 2023. <https://www.fluentcpp.com/2021/03/12/cpp-fold-expressions/>.
 87. Jonathan Boccaro, “What C++ Fold Expressions Can Bring to Your Code,” March 19, 2021. Accessed April 18, 2023. <https://www.fluentcpp.com/2021/03/19/what-c-fold-expressions-can-bring-to-your-code/>.
 88. Jonathan Muller, “Nifty Fold Expression Tricks,” May 5, 2020. Accessed April 18, 2023. <https://www.foonathan.net/2020/05/fold-tricks/>.

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`.



The C++ Core Guidelines recommend using **variadic function templates** for arguments of **mixed types**⁸⁹ and **initializer_lists** for functions that receive variable numbers of arguments of the **same type**.⁹⁰ However, **fold expressions cannot be applied to initializer_lists**.

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 expressions⁹¹ and
- *initialValue* represents a starting value for a binary fold expression. For example, a sum might start with 0, and a product might start with 1.

There are **four fold-expression types**—the parentheses are required in each:

- A **unary left fold** has one binary operator with the parameter pack expansion (...) as the **left operand** and the *pack* as the right operand:

(... *op* *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* *op* ...)

- A **binary left fold** has two binary operators, which must be the same. The *initialValue* is to the **left of the first operator**, the parameter pack expansion (...) is between the operators, and *pack* is to the right of the second operator:

(*initialValue* *op* ... *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 (...) is between the operators, and *pack* is to the left of the first operator:

(*pack* *op* ... *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

89. C++ Core Guidelines, “T.100: Use Variadic Templates When You Need a Function That Takes a Variable Number of Arguments of a Variety of Types.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic>.

90. C++ Core Guidelines, “T.103: Don’t Use Variadic Templates for Homogeneous Argument Lists.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic-not>.

91. “Fold Expression.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/fold>.

demonstrates unary-left-fold and unary-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** to add the items in its parameter pack.
- Lines 16–19 define **unaryLeftSubtract**, which uses a **unary left fold** to subtract the items in its parameter pack.
- Lines 21–24 define **unaryRightSubtract**, which uses a **unary right fold** to subtract the items in its parameter pack.

```
1 // fig15_15.cpp
2 // Unary fold expressions.
3 #include <format>
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     << std::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     << std::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 }
```

Fig. 15.15 | Unary fold expressions. (Part I of 2.)

```

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. (Part 2 of 2.)

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 line 35 call

```
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

```
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;** otherwise, a compilation error occurs. 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()**, meaning 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.

```
1 // fig15_16.cpp
2 // Binary fold expressions.
3 #include <format>
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
```

Fig. 15.16 | Binary fold expressions.

```

26 int main() {
27     std::cout << "Binary left and right fold with addition:\n"
28     << std::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:\n"
35     << std::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
36                 "binaryLeftSubtract(): ", binaryLeftSubtract(),
37                 "binaryLeftSubtract(1, 2, 3, 4): ", binaryLeftSubtract(1, 2, 3, 4),
38                 "binaryRightSubtract(): ", binaryRightSubtract(),
39                 "binaryRightSubtract(1, 2, 3, 4): ", binaryRightSubtract(1, 2, 3, 4));
40
41 }
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 7 executes the expression on the left of the comma operator

```
(std::cout << items << "\n")
```

once for each item in the parameter pack `items`.

```

1 // fig15_17.cpp
2 // Repeating a task using the comma operator and fold expressions.
3 #include <iostream>
4
```

Fig. 15.17 | Repeating a task using the comma operator and fold expressions. (Part I of 2.)

```

5  template <typename... Items>
6  void printItems(Items... items) {
7      ((std::cout << items << "\n"), ...); // unary right fold
8  }
9
10 int main() {
11     std::cout << "printItems(1, 2.2, \"hello\"): \n";
12     printItems(1, 2.2, "hello");
13 }
```

```

printItems(1, 2.2, "hello"):
1
2.2
hello
```

Fig. 15.17 | Repeating a task using the comma operator and fold expressions. (Part 2 of 2.)

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 **pre-defined concept** `std::same_as` to check that all the elements of a parameter pack have the same type as another type argument.

```

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... Us>
9 concept SameTypeElements = (std::same_as<T, Us> && ...);
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 }
```

Fig. 15.18 | Constraining a variadic-function-template parameter pack to elements of the same type. (Part 1 of 2.)

```

fig15_18.cpp:23:4: error: no matching function for call to 'foldingSum'
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 'SameTypeElements<int, double,
std::basic_string<char>>' evaluated to false
    requires SameTypeElements<FirstItem, RemainingItems...>
        ^
fig15_18.cpp:9:34: note: because 'std::same_as<int, double>' evaluated to
false
concept SameTypeElements = (std::same_as<T, Us> && ...);
    ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, double>' evaluated to
false
    = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
        ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, double>' evaluated to false
    concept __same_as = std::is_same_v<_Tp, _Up>;
        ^
fig15_18.cpp:9:34: note: and 'std::same_as<int, std::basic_string<char>>'
evaluated to false
concept SameTypeElements = (std::same_as<T, Us> && ...);
    ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, std::basic_string<char>
>' evaluated to false
    = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
        ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, std::basic_string<char>>'
evaluated to false
    concept __same_as = std::is_same_v<_Tp, _Up>;
        ^
1 error generated.

```

Fig. 15.18 | Constraining a variadic-function-template parameter pack to elements of the same type. (Part 2 of 2.)

Concepts

C Custom Concept for a Variadic Template

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, Us>
```

compares the type `T` to one type from the parameter pack `Us`. This constraint is in a fold expression, so the **parameter pack expansion** (...) applies `std::same_as<T, Us>` once for each type from the parameter pack `Us`, comparing every type in `Us` to the type `T`. If every item in the parameter pack `Us` 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`

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 `Us`. Of course, `double` and `string` are not `int`, so each use of the concept `std::same_as<T, Us>` fails. Figure 15.18 shows the **Clang C++ compiler error messages**. We highlighted key messages in bold and added vertical spacing for readability. Clang indicates



```
error: no matching function for call to 'foldingSum'
```

and

```
note: candidate template ignored: constraints not satisfied
```

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 'std::same_as<int, double>' evaluated to false
```

```
note: and 'std::same_as<int, std::basic_string<char> >' evaluated to false
```

15.13 Template Metaprogramming

As we've mentioned, a goal of modern C++ is doing more 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** introduced so far are crucial template-metaprogramming capabilities.

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 manageable template-metaprogramming examples. We provide many footnote citations containing 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**,^{92,93} so anything that can be computed, 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.^{94,95} The program did not compile, but 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

<http://www.erwin-unruh.de/primorig.html>

Here are some selected lines from those error messages. The specializations of his class template D showing the first several prime numbers are highlighted in bold:

```
| Type `enum{}' can't be converted to type `D<2>' ("primes.cpp",L2/C25).
| Type `enum{}' can't be converted to type `D<3>' ("primes.cpp",L2/C25).
| Type `enum{}' can't be converted to type `D<5>' ("primes.cpp",L2/C25).
| Type `enum{}' can't be converted to type `D<7>' ("primes.cpp",L2/C25).
```

15.13.2 Computing Values at Compile-Time



The goal of compile-time calculations is to optimize a program's runtime performance.⁹⁶

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 5.16. The C++ Core Guidelines recommend using `constexpr` functions to compute values at compile-time⁹⁷—such functions use traditional C++ syntax.

92. Todd Veldhuizen, "C++ Templates Are Turing Complete," 2003. Accessed April 18, 2023. <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670>.

93. "Template Metaprogramming." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Template_metaprogramming.

94. Erwin Unruh, "Prime Number Computation," 1994. ANSI X3J16-94-0075/ISO WG21-4-62.

95. Rainer Grimm, "C++ Core Guidelines: Rules for Template Metaprogramming," January 7, 2019. Accessed April 18, 2023. <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.

96. "Template Metaprogramming." Wikipedia.

97. C++ Core Guidelines, "T.123: Use `constexpr` Functions to Compute Values at Compile Time." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-fct>.



```
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 instantiating
40     // the template and accessing its static value member
41     std::cout << "CALCULATING FACTORIALS AT COMPILE TIME "
42         << "WITH 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 }
```

Fig. 15.19 | Calculating factorials at compile-time. (Part I of 2.)

```

50 // calling the recursive constexpr function recursiveFactorial
51 std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
52     << "WITH 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 iterativeFactorial
61 std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
62     << "WITH 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

```

Fig. 15.19 | Calculating factorials at compile-time. (Part 2 of 2.)

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 typically named **value**. The class template uses its template arguments to compute a 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 conventions⁹⁸ used throughout the C++ standard library and in metaprogramming in general.

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 metafunctions, 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 instantiating its template, which probably feels weird to you. For example, to calculate the factorial of 3, you’d use

```
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 **metafunction 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 1.⁹⁹

98. Jody Hagins, “Template Metaprogramming: Type Traits (Part 1 of 2),” YouTube video, September 22, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=tAVWcjIF6o>.

99. Section 5.16 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 43 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. 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.

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 perform these calculations at compile-time.

Functional Programming

None of what the compiler does during template specialization modifies variable values after they're initialized.¹⁰⁰ 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.

¹⁰⁰“Template Metaprogramming.” Wikipedia.

Using `constexpr` Functions to Perform Compile-Time Calculations

As you can see, using metafunctions to calculate values at compile-time is more complex than 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.
- In C++20, you can declare a function `constexpr` (rather than `constexpr`) to indicate that it can be called only at compile-time to produce a constant.

Many computations performed with metafunctions are easier to implement using traditional functions that are declared `constexpr` or `constexpr`. 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++.”¹⁰¹ They’ve proposed various `constexpr` enhancements to simplify other aspects of metaprogramming in future C++ versions.

ASSE

Lines 18–25 and 28–36 define the `recursiveFactorial` and `iterativeFactorial` functions using the algorithms from Section 5.16 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`

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. Figure 15.20 reimplements that example using a single `customDistance` function template, showing how you could implement similar functionality before concepts. Inside the function, we use a **compile-time if statement**—`constexpr if`—with `std::is_base_of_v` (the shorthand for accessing type trait `std::is_base_of`’s `value` member) to decide whether to generate the $O(1)$ or $O(n)$ distance calculation based on whether the function’s iterator arguments are

- random-access iterators (or the derived iterator category contiguous iterators) that can be used to perform an $O(1)$ distance calculation or
- lesser iterators for which we’ll perform the $O(n)$ distance calculation.

```
1 // fig15_20.cpp
2 // Implementing customDistance using template metaprogramming.
3 #include <array>
4 #include <iostream>
```

Fig. 15.20 | Implementing `customDistance` using template metaprogramming. (Part 1 of 2.)

101. Louis Dionne, “Metaprogramming By Design, Not By Accident,” June 18, 2017. Accessed April 18, 2023. <https://wg21.link/p0425r0>.

```

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 <std::input_iterator Iterator>
13 auto customDistance(Iterator begin, Iterator end) {
14     // for random-access iterators, subtract the iterators
15     if constexpr (std::is_base_of_v<std::random_access_iterator_tag,
16                   typename Iterator::iterator_category>) {
17
18         std::cout << "customDistance with random-access iterators\n";
19         return end - begin; // O(1) operation for random-access iterators
20     }
21     else { // for all other iterators
22         std::cout << "customDistance with non-random-access iterators\n";
23         std::ptrdiff_t count{0};
24
25         // increment from begin to end and count number of iterations;
26         // O(n) operation for non-random-access iterators
27         for (auto iter{begin}; iter != end; ++iter) {
28             ++count;
29         }
30
31         return count;
32     }
33 }
34
35 int main() {
36     const std::array ints1{1, 2, 3, 4, 5}; // has random-access iterators
37     const std::list ints2{1, 2, 3}; // has bidirectional iterators
38
39     auto result1{customDistance(ints1.begin(), ints1.end())};
40     std::cout << "ints1 number of elements: " << result1 << "\n";
41     auto result2{customDistance(ints2.begin(), ints2.end())};
42     std::cout << "ints2 number of elements: " << result2 << "\n";
43 }

```

```

customDistance with random-access iterators
ints1 number of elements: 5
customDistance with non-random-access iterators
ints2 number of elements: 3

```

Fig. 15.20 | Implementing `customDistance` using template metaprogramming. (Part 2 of 2.)

Lines 12–33 define function template `customDistance`. Since this function requires **at least input iterators** to perform its task, line 12 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 15). This statement's condition must evaluate at compile-time to a `bool`.

Lines 15–16 use `std::is_base_of_v` to compare two types:

- `std::random_access_iterator_tag` and
- `Iterator::iterator_category`.

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`
- `contiguous_iterator_tag`

These are implemented as a class hierarchy.¹⁰² In line 16, the expression

```
Iterator::iterator_category
```

gets the iterator tag from the argument's iterator type, which is `std::is_base_of_v`, then compares it to its first type argument

```
std::random_access_iterator_tag
```

to determine whether they're either the same type or `std::random_access_iterator_tag` is a base class of `Iterator::iterator_category`. The result is `true` or `false`. If `true`, the compiler instantiates the code in the `if constexpr` body (lines 18–19), which is optimized for `random-access iterators` to perform the $O(1)$ calculation. Otherwise, the compiler instantiates the `else`'s body (lines 22–31), which uses the $O(n)$ approach that works for all other iterator types. The `g++` and `clang++ std::array` implementations use pointers as iterators. These will not work with this `customDistance` implementation, however the concept-based overloading example in Section 15.6.4 does work with pointers as iterators.

15.13.4 Type Metafunctions

Type metafunctions are frequently 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. In Fig. 15.21, 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.



¹⁰²“`std::input_iterator_tag`, `std::output_iterator_tag`, `std::forward_iterator_tag`, `std::bidirectional_iterator_tag`, `std::random_access_iterator_tag`, `std::contiguous_iterator_tag`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/iterator/iterator_tags.

```

1 // fig15_21.cpp
2 // Adding and removing type attributes with type metafunctions.
3 #include <format>
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 << std::format("{}\n{}{}\n{}{}\n{}\n",
27         "ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION",
28         "std::is_same_v<const int, my_add_const<int>::type>: ",
29         std::is_same_v<const int, my_add_const<int>::type>,
30         "std::is_same_v<int* const, my_add_const<int*>::type>: ",
31         std::is_same_v<int* const, my_add_const<int*>::type>);

32     std::cout << std::format("{}\n{}{}\n{}{}\n{}\n",
33         "REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION",
34         "std::is_same_v<int, my_remove_ptr<int>::type>: ",
35         std::is_same_v<int, my_remove_ptr<int>::type>,
36         "std::is_same_v<int, my_remove_ptr<int*>::type>: ",
37         std::is_same_v<int, my_remove_ptr<int*>::type>);

38     std::cout << std::format("{}\n{}{}\n{}{}\n{}\n",
39         "ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS",
40         "std::is_same_v<int&, std::add_lvalue_reference<int>::type>: ",
41         std::is_same_v<int&, std::add_lvalue_reference<int>::type>,
42         "std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: ",
43         std::is_same_v<int&&, std::add_rvalue_reference<int>::type>);

44     std::cout << std::format("{}\n{}{}\n{}{}\n{}\n",
45         "REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS",
46         "std::is_same_v<int, std::remove_reference<int>::type>: ",
47         std::is_same_v<int, std::remove_reference<int>::type>,
48         "std::is_same_v<int, std::remove_reference<int&>::type>: ",
49         std::is_same_v<int, std::remove_reference<int&>::type>,
50         "std::is_same_v<int, std::remove_reference<int&&>::type>: ",
51         std::is_same_v<int, std::remove_reference<int&&>::type>,
52         std::is_same_v<int, std::remove_reference<int&&>::type>,

```

Fig. 15.21 | Compile-time type manipulation. (Part I of 2.)

```

53     "std::is_same_v<int, std::remove_reference<int&&>::type>: ",
54     std::is_same_v<int, std::remove_reference<int&&>::type>);
55 }

```

```

ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<const int, my_add_const<int>::type>: true
std::is_same_v<int* const, my_add_const<int*>::type>: true

REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<int, my_remove_ptr<int>::type>: true
std::is_same_v<int, my_remove_ptr<int*>::type>: true

ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int&, std::add_lvalue_reference<int>::type>: true
std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: true

REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int, std::remove_reference<int>::type>: true
std::is_same_v<int, std::remove_reference<int&>::type>: true
std::is_same_v<int, std::remove_reference<int&&>::type>: true

```

Fig. 15.21 | Compile-time type manipulation. (Part 2 of 2.)

Adding const to a Type

Lines 8–11 implement a type metafunction named `my_add_const`—a simplified version of the `std::add_const` metafunction from header `<type_traits>`. By convention, a type metafunction must have a `public` member named `type`. When `my_add_const` is instantiated with a type, the compiler defines the type alias in line 10, which precedes the type with `const`.

Lines 26–31 in `main` demonstrate the `my_add_const` metafunction. Line 29

```
my_add_const<int>::type
```

instantiates the template with the non-const type `int`. To confirm that `const` was added to `int`, we use `std::is_same_v` (the shorthand for accessing `std::is_same`'s `value` member) to compare the type `const int` to the type returned by the preceding expression. They are the same, so the result is `true`, as shown in the output. Similarly, line 31 shows that the `my_add_const` metafunction also works with pointer types.

Removing * from a Pointer Type

Lines 14–17 and 20–23 implement a custom type metafunction named `my_remove_ptr` that mimics the `std::remove_pointer` metafunction from the `<type_traits>` header to show how that metafunction could be implemented. This requires two metafunction 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 any type. For a type that does not include an `*` to indicate a pointer, this metafunction returns the type used to specialize the template.

- The metafunction class template in lines 20–23 matches only pointer types. For this purpose, we define a **partial template specialization**.¹⁰³ 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`.

Lines 36 and 38 in `main` demonstrate our `custom my_remove_ptr` type metafunctions. Let's consider the compiler's matching process for invoking them.

Instantiating `my_remove_pointer` for Non-Pointer Types

The expression in line 36

```
my_remove_ptr<int>::type
```

instantiates the template with the non-pointer type `int`. The compiler must decide which definition of `my_remove_ptr` matches the expression. Since there is no `*` to indicate a pointer in the type `int`, this expression 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 `std::is_same_v` to compare type `int` to the type returned by the preceding expression. They are the same, so the result is `true`.

Instantiating `my_remove_pointer` for Pointer Types

The expression in line 38

```
my_remove_ptr<int*>::type
```

instantiates 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_v` to compare `int` to the type returned by the preceding expression. They are the same, so the result is `true`.

Adding `lvalue` and `rvalue` References to a Type

The predefined type metafunctions that modify types work similarly to `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

¹⁰³Inbal Levi, “Exploration of C++20 Metaprogramming,” September 29, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=XgrjybKaIV8>.

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_v` 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 references from a type. We instantiate `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_v` to compare `int` to the type returned by each expression in lines 50, 52 and 54. They are the same, so the result is `true` in each case.

15.14 Wrap-Up

This chapter 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 when you instantiate templates.

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 also introduced 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 introduce modules—one of C++20’s “big four” features (along with ranges, concepts and coroutines), which provide a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details.

Self-Review Exercises

15.1 State which of the following are *true* and which are *false*. If *false*, explain why.

- a) A modern C++ theme is doing more at compile-time for better type checking and runtime performance.
- b) A template enables runtime polymorphism by specifying capabilities generically, then letting the compiler instantiate the template, generating type-specific code specializations on demand.
- c) C++20 Concepts can constrain the types passed to a function template and prevent the compiler from attempting to instantiate the template.
- d) Concepts specify template requirements explicitly in code, enabling the compiler to determine that a type is incompatible with a template after instantiating it.
- e) The C++ Core Guidelines recommend specifying custom C++20 concepts for every template parameter.
- f) Anywhere you can use `auto` in your code, you can precede it with a concept name to constrain the allowed types.
- g) When multiple function templates satisfy a function call, the compiler calls the least constrained version.
- h) `static_assert` was developed to add run-time assertion support for reporting incorrect usage of template libraries.
- i) The C++ Core Guidelines recommend using templates to implement any class representing a container of values or range of values.
- j) The compiler needs the complete definition where a template is used to instantiate it.
- k) You can use compile-time recursion to process variadic parameter packs.
- l) Template metaprogramming (TMP) enables the compiler to manipulate types, perform compile-time computations and generate optimized code.

15.2 Fill in the blanks in each of the following:

- a) The compiler uses class templates to generate related classes—this is known as _____.
_____.
- b) To use a type with a template, the type must _____.
_____.
- c) Compilers generate definitions only for _____.
_____.
- d) C++20 abbreviated function templates enable you to define a function template without the template header by using the _____ keyword as the parameter type.
_____.
- e) Concept-based overloading enables the compiler to choose between overloads _____, even if the overloads have the same signature.
_____.
- f) All iterators must support default construction, copy construction, copy assignment, destruction and _____.
_____.
- g) A range-based `for` requires only _____ iterators, so it works with any iterable object.
_____.
- h) Creating a `tuple` object is called _____.
_____.
- i) _____ provide a convenient notation for applying a binary operator to all the elements in a variadic template's parameter pack.
_____.

- j) Todd Veldhuizen proved that C++ templates are _____ so anything that can be computed, can be computed at compile-time with C++ template metaprogramming.
- k) You “call” a metafunction by _____.
- l) There are no _____ variables in template metaprograms. This is a hallmark of functional programming.

Answers to Self-Review Exercises

15.1 a) True. b) False. This is how a template enables compile-time polymorphism. c) True. d) False. Actually, they enable the compiler to determine that a type is incompatible with a template before instantiating it—leading to fewer, more precise error messages, as well as potential compile-time performance improvements. e) False. Actually, the C++ Core Guidelines recommend specifying concepts for every template parameter using the standard’s predefined concepts, if possible. f) True. g) False. Actually, when multiple function templates satisfy a function call, the compiler calls the most constrained version. h) False. Actually, `static_assert` was developed to add compile-time assertion support for reporting incorrect usage of template libraries. i) True. j) True. k) True. l) True.

15.2 a) instantiating the class template. b) satisfy the template’s requirements. c) the portions of a template used in your code. d) `auto`. e) based on the concepts used to constrain parameters. f) swapping. g) input. h) packing a `tuple`. i) Fold expressions. j) Turing complete. k) instantiating its template. l) mutable.

Exercises

- 15.3** State which of the following are *true* and which are *false*. If *false*, explain why.
- a) The compiler checks concepts before instantiating templates’ bodies, often resulting in fewer, more-precise error messages.
 - b) One `Stack` class template could become the basis for creating many `Stack` class-template specializations, such as “`Stack of doubles`,” “`Stack of Employees`,” “`Stack of Bills`” or “`Stack of ActivationRecords`.”
 - c) Concepts can be applied to any template parameter and to any use of `auto`.
 - d) Each constraint specified by a C++20 Concept is a run-time predicate expression.
 - e) When functions perform syntactically identical operations on different types, they can be expressed more compactly using overloaded functions.
 - f) With C++20, you can use concepts in function templates to select overloads based on type requirements.
 - g) When `static_assert`’s argument is `false`, the compiler outputs an error message that tells you what and where the error is. If the argument is `true`, the compiler does not output any messages—it simply continues compiling the code.
 - h) The C++ Core Guidelines recommend using templates to implement any class representing a container of values or range of values.
 - i) Most standard library algorithms operate on ranges of container elements.
 - j) The C++ Core Guidelines recommend using `initializer_lists` for arguments of mixed types and variadic function templates for functions that receive variable numbers of arguments of the same type.

- k) 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 right-most expression. You can combine the comma operator with fold expressions to repeatedly perform tasks for every item in a parameter pack.
- l) Always prefer the `<type_traits>` header's predefined type traits rather than implementing your own.

15.4 Fill in the blanks in each of the following:

- a) Instantiating templates enables _____ (static) polymorphism.
- b) Class templates encourage software reuse by enabling the compiler to instantiate many type-specific _____ from a single class template.
- c) 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 _____.
- d) When you call a function, the compiler uses its overload-resolution rules and a technique called argument-dependent lookup (ADL) to locate all the function definitions that might satisfy the function call. Together, these are known as the _____.
- e) You can define custom algorithms capable of operating on any container that supports your algorithm's _____ requirements.
- f) You can implement your iterator classes using the _____, letting the compiler generate the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor special member functions.
- g) If you have a non-`const` object that you want to treat as `const`, you can create a `const` view of the object by passing it to the _____ function.
- h) You can use _____ to define groups of related variables.
- i) You can pack a tuple with the `<tuple>` header's _____ function.
- j) The goal of compile-time calculations is to _____.
- k) The C++ Core Guidelines recommend using _____ functions to compute values at compile time—such functions use traditional C++ syntax.
- l) In C++20, you can declare a function _____ to indicate that it can be called only at compile-time to produce a constant.

Discussion Exercises

15.5 (*Discussion*) Distinguish between the terms “function template” and “function-template specialization.”

15.6 (*Discussion*) Explain which is more like a stencil—a class template or a class-template specialization?

15.7 (*Discussion*) Why is it appropriate to refer to a class template as a parameterized type?

15.8 (*Discussion*) Why are a class template's member functions also defined in the header?

15.9 (*Discussion*) Why might you use a nontype parameter with a class template for a container such as an array or stack?

15.10 (*Discussion*) Explain why, if you wish to use a function template's parameter names in a constraint, you must use a trailing `requires` clause.

15.11 (*Discussion*) When determining which 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. What happens if there's more than one best match?

15.12 (*Discussion*) Describe how function templates may be overloaded.

15.13 (*Discussion*) Some algorithms can be optimized for specific iterator types. For example, explain how `std::distance` can be implemented as an $O(1)$ operation for random-access iterators.

15.14 (*Discussion*) Variadic templates accept any number of arguments. Why do they simplify template programming?

Code Exercises

15.15 (*Function Template*) Create a `sumRange` function template that receives a range of values, sums them and returns the result. Use a range-based `for` statement and the `+=` operator to perform your calculations. Test your code with a container of `ints`, a container of `doubles` and a container of `strings`.

15.16 (*Abbreviated Function Template*) Modify your solution to Exercise 15.15 to reimplement `sumRange` as an abbreviated function template.

15.17 (*Concept Constrained Abbreviated Function Template*) Modify your solution to Exercise 15.16 to use C++20 concepts to constrain `sumRange`'s parameter to a range with elements that are integers or floating-point values.

15.18 (*Concept Constrained Abbreviated Function Templates*) Reimplement Fig. 15.8's `customDistance` function templates as concept-constrained abbreviated function templates.

15.19 (*Operator Overloads in Templates*) Write a simple function template for predicate function `isEqualTo` that compares its two arguments of the same type with the equality operator (`==`) and returns `true` if they are equal and `false` otherwise. Use this function template in a program that calls `isEqualTo` only with a variety of fundamental types. Now write a separate version of the program that calls `isEqualTo` with a user-defined class type but does not overload the equality operator. What happens when you attempt to run this program? Next, define an overloaded `operator==`. Now, what happens when you attempt to run this program?

15.20 (*Trailing requires Clause and an Ad-Hoc Constraint*) Create an overloaded binary `*` operator that requires a `string` as its first argument and an `int` as its second argument. The operator function should return a new `string` containing the first argument repeated the number of times specified by the second argument. For example, given

```
std::string s{"ha"};
```

the expression

```
s * 4
```

would repeat "ha" four times and return a string containing "hahahaha".

Next, modify Fig. 15.5’s `multiply` function template to use two type parameters named `T` and `U` so the function’s arguments can have the same or different types. Change the return type to `auto`, so the compiler can determine the function’s return type from the function’s return value. Rather than requiring the type arguments to have integral or floating-point types, use a trailing `requires` clause and an ad-hoc constraint to indicate that the parameters `first` and `second` must be able to be used in an expression of the form

```
first * second
```

Use your updated `multiply` function template in an application that multiplies:

- a) two `ints`.
- b) two `doubles`.
- c) an `int` and a `double`.
- d) a `string` and an `int`.

15.21 (Custom Concept) Define a custom concept named `product` that uses a simple requirement (Section 15.6.5) to specify that arguments of its two type parameters `first` and `second` must support the expression

```
first * second
```

Modify your `multiply` function template from Section 15.20 to replace its ad-hoc constraint with your custom concept named `product`, then re-test the application.

15.22 (MyArray Class Template) Reimplement Section 11.6’s `MyArray` class as a class template. Modify the main program to use this `MyArray` class template specialized for type `int`, then demonstrate that its features also work for a `MyArray` specialized for type `double`.

15.23 (tuples of Student Grades) Write an application that represents an instructor’s grade book as an array of `tuples`, each consisting of a student’s first name (a `string`) and three `int` values for the student’s grades on three exams. Using the array’s data and the `tuple` techniques presented in Section 15.12.1, calculate each student’s average on the three exams and calculate the total class average. Specify the array’s data at compile time with the following student data:

- Susan: 92, 85, 100
- Eduardo: 83, 95, 79
- Azizi: 91, 89, 82
- Pantipa: 97, 91, 92

Given this data, the program should produce the following output:

```
Average for Susan is 92.33
Average for Eduardo is 85.67
Average for Azizi is 87.33
Average for Pantipa is 93.33
Class's average is: 89.67
```

15.24 (Calculating Students’ Averages with Fold Expressions) Modify your solution to Exercise 15.23 to use fold expressions and `sizeof...` to calculate each student’s average.

15.25 (Computing Values at Compile-Time with Metaprogramming) Using the techniques in Section 15.13.2, create a template metafunction, a `constexpr` recursive function

and a `constexpr` iterative function to calculate the sum of the squares of the `int` values from 1 through the argument value.

15.26 (Project: MyArray Container Class Template with Custom Random-Access Iterators) As we mentioned in Section 15.9, the iterators for our `MyArray` container class template were inspired by Microsoft's `std::array` iterator implementation. Study Microsoft's implementation of their iterator classes at:

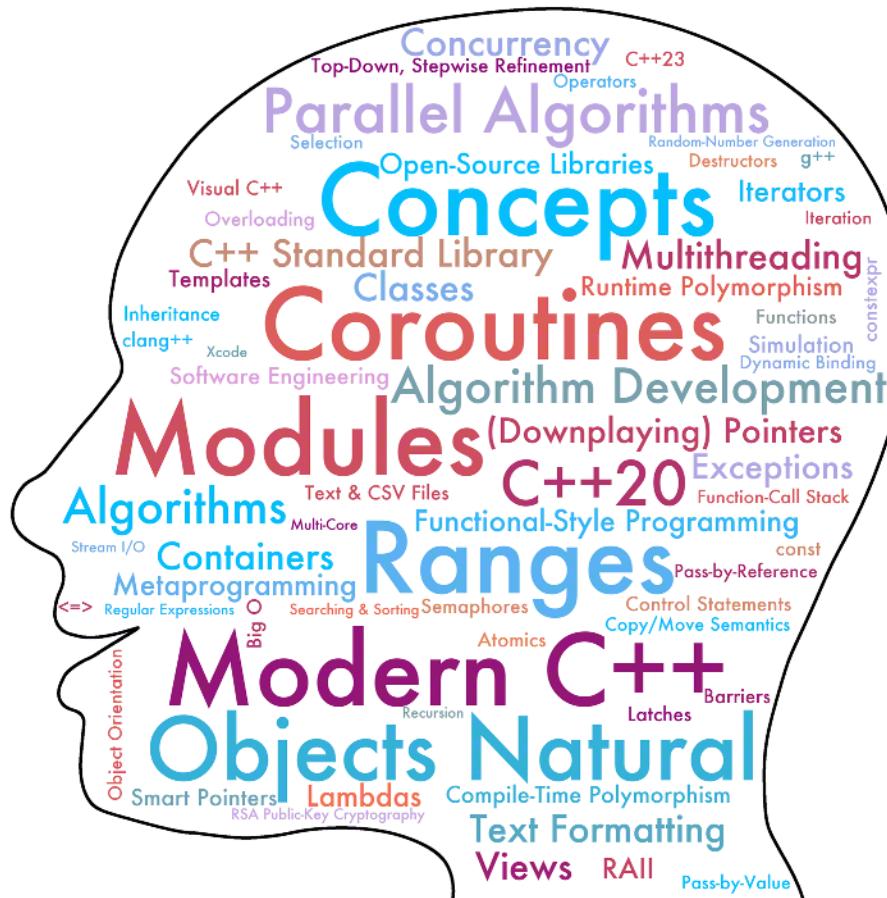
<https://github.com/microsoft/STL/blob/main/stl/inc/array>

The enhance Section 15.9's `ConstIterator` and `Iterator` classes to support all the operations required for random-access iterators. Test your updated iterators by using an object of class `MyArray` with

- `std::ranges::shuffle` to randomly order the `MyArray`'s elements, then
- `std::ranges::sort` to sort the `MyArray`'s elements into ascending order.

This page intentionally left blank

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.
- Define a module's primary interface unit.
- `export` declarations from a module for use in other translation units.
- `import` modules to use their exported declarations.
- Separate a module's interface from its implementation via a `:private` module fragment or module implementation unit.
- See the compilation errors when attempting to use non-exported module items.
- Organize modules into logical components via partitions.
- Divide a module into "submodules" so 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	16.9 Partitions
16.2 Compilation and Linking Before C++20	16.9.1 Example: Module Interface Partition Units
16.3 Advantages and Goals of Modules	16.9.2 Module Implementation Partition Units
16.4 Example: Transitioning to Modules—Header Units	16.9.3 Example: “Submodules” vs. Partitions
16.5 Modules Can Reduce Translation Unit Sizes and Compilation Times	16.10 Additional Modules Examples
16.6 Example: Creating and Using a Module	16.10.1 Example: Importing the C++ Standard Library as Modules
16.6.1 <code>module</code> Declaration for a Module Interface Unit	16.10.2 Example: Cyclic Dependencies Are Not Allowed
16.6.2 Exporting a Declaration	16.10.3 Example: <code>imports</code> Are Not Transitive
16.6.3 Exporting a Group of Declarations	16.10.4 Example: Visibility vs. Reachability
16.6.4 Exporting a <code>namespace</code>	16.11 Migrating Code to Modules
16.6.5 Exporting a <code>namespace</code> Member	16.12 Future of Modules and Modules Tooling
16.6.6 Importing a Module to Use Its Exported Declarations	16.13 Wrap-Up
16.6.7 Example: Attempting to Access Non-Exported Module Contents	Exercises
16.7 Global Module Fragment	Appendix: Modules Videos
16.8 Separating Interface from Implementation	Bibliography
16.8.1 Example: Module Implementation Units	Appendix: Modules Articles
16.8.2 Example: Modularizing a Class	Bibliography
16.8.3 <code>:private</code> Module Fragment	Appendix: Modules Glossary

16.1 Introduction

 **Modules**—one of C++20’s “big four” features (along with ranges, concepts and coroutines)—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.

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 aim to 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

1. Daveed Vandevoorde, “Modules in C++ (Revision 6),” January 11, 2012. Accessed April 18, 2023. <https://wg21.link/n3347>.
2. Gabriel Dos Reis, “Programming in the Large with C++ 20: Meeting C++ 2020 Keynote,” December 11, 2020. Accessed April 18, 2023. <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 April 18, 2023. <https://arxiv.org/pdf/2004.06507.pdf>.
4. Bjarne Stroustrup, “Modules and Macros,” February 11, 2018. Accessed April 18, 2023. <https://wg21.link/p0955r0>.

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.

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. We present the required compilation commands for each example and indicate which compiler(s), if any, did not support a given feature.⁵ As you'll see, Visual C++ can compile modules for you. We provide a file named `compilation_commands.txt` with this chapter's examples. It contains each example's `g++` and `clang++` compilation commands, so you can conveniently copy and paste them.

Docker Containers for g++ 13.1 and clang++ 16

To test our modules examples in the most current `g++` and `clang++` versions, we used the following free Docker containers from <https://hub.docker.com>—if you already followed the instructions in the Before You Begin section to install Docker and these Docker containers, you can skip these commands:

- For `g++`, we used the official GNU GCC container's latest version (13.1):

```
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 a widely downloaded one containing `clang++` version 16:

```
docker pull teeks99/clang-ubuntu:16
```

C++ Compilation and Linking Before C++20

Section 16.2 discusses the traditional C++ compilation and linking process and various problems with that model.

Advantages and Goals of Modules

Section 16.3 points 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 discusses and provides references for some C++23 module 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.

5. The compilation steps might change. We'll post updates at <https://deitel.com/cpphtp11>.

16.2 Compilation and Linking Before 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**.⁶
- 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 and was inherited into C++ from C.⁷ If you're not familiar with the preprocessor, you might want to read the Preprocessor appendix at <https://deitel.com/cpphtp11>, 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++. To motivate C++20 modules, C++ creator Bjarne Stroustrup calls out three preprocessor problems:⁸



- One header's contents can affect any subsequent `#included` header, so **the order of `#includes` is important** and can cause subtle errors.
- A given C++ entity can have **different declarations in multiple translation units**. The compiler and linker do not always report such problems.
- **Reprocessing the same `#included` content slows down compilation**, particularly in large programs where a header can be included dozens or even hundreds of times. In each **translation unit**, the preprocessor will insert an `#included` header's contents, possibly causing the same code to be compiled repeatedly in many translation units. Eliminating this reprocessing by **importing headers as header units** (Section 16.4) can significantly improve compilation times in large codebases.

Other preprocessor problems include:⁹

- Definitions in headers can violate C++'s **One Definition Rule (ODR)**, which 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."¹⁰

6. "Translation Unit (Programming)." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. [https://en.wikipedia.org/wiki/Translation_unit_\(programming\)](https://en.wikipedia.org/wiki/Translation_unit_(programming)).

7. Gabriel Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer," October 5, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=tjSuK0z5HK4>.

8. Stroustrup, "Thriving in a Crowded and Changing World."

9. Bryce Adelstein Lelbach, "Modules Are Coming," May 1, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=yee9i2rUF3s>.

10. "C++20 Standard: 6 Basics—6.3 One-Definition Rule." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/basic.def.odr>.

- Headers do not offer encapsulation—everything in a header is available to the translation unit that `#includes` the header.
- Accidental cyclic dependencies between headers cause compilation errors and other problems.¹¹
- Headers often define macros—`#define` preprocessor directives that create constants represented as symbols (such as `#define SIZE 10`), as well as function-like operations (such as `#define SQUARE(x) ((x) * (x))`).¹² The preprocessor handles macros via text substitutions, so the compiler cannot check macros' syntax and cannot report multiple-definition errors if two or more headers define the same macro names. 

16.3 Advantages and Goals of Modules

Some benefits of using modules include:^{13,14,15,16,17}

- Better organization and componentization of large codebases. 
- Smaller translation unit sizes.
- Reduced compilation times.¹⁸ 
- Eliminating repetitive `#include` processing—a compiled module does not have to be reprocessed 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.

11. “Circular Dependency.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Circular_dependency.

12. The all capital letters naming for preprocessor constants (like `SIZE`) and macros (like `SQUARE`) is a convention that some programmers prefer.

13. Corentin Jabot, “What Do We Want from a Modularized Standard Library?,” May 16, 2020. Accessed April 18, 2023. <https://wg21.link/p2172r0>.

14. Gabriel Dos Reis and Pavel Curtis, “Modules, Componentization, and Transition,” October 5, 2015. Accessed April 18, 2023. <https://wg21.link/p0141r0>.

15. Daniela Engert, “Modules: The Beginner’s Guide,” May 2, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

16. Rainer Grimm, “C++20: The Advantages of Modules,” May 10, 2020. Accessed April 18, 2023. <https://www.modernescpp.com/index.php/cpp20-modules>.

17. Dmitry Guzev, “A Few Words on C++ Modules,” January 8, 2018. Accessed April 18, 2023. <https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64>.

18. Rene Rivera, “Are Modules Fast? (Revision 1),” March 6 2019, Accessed April 18, 2023. <https://wg21.link/p1441r1>.

Cons of Modules

Some disadvantages of using modules include:^{19,20}

- Modules support is incomplete in some C++ compilers at the time of this writing.
- Existing codebases will need to be modified to fully benefit from modules.
- Modules do not solve C++ problems with respect to packaging and distributing software conveniently, as we can do with package managers in several other popular languages.
- Compiled modules have compiler-specific aspects that are not portable across compilers and platforms, so you still need to distribute modules as source code.
- Modules uptake will initially 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.
- Few recommendations and guidelines for modules currently exist. For example, the C++ Core Guidelines have not yet been updated for modules.

16.4 Example: Transitioning to Modules—Header Units



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 platform-specific object-code files**.

It will take time for library developers to modularize their libraries. Some might never be modularized. As a transitional step from the preprocessor to modules, you can **import** (most) existing headers from the C++ standard library,²¹ as shown in line 3 of Fig. 16.1. Doing so treats that header as a **header unit**.

```

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!

Fig. 16.1 | Importing a standard library header as a header unit.

19. Steve Downey, “Writing a C++20 Module,” July 5, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=A04piAqV9mg>.
20. “C++ Modules Might Be Dead-on-Arrival,” January 27, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/01/27/modules-doa.html>.
21. “C++20 Standard: 10 Modules—10.3 Import Declaration.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.import>.

How Header Units Differ from Header Files

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.²² Header units are similar to **precompiled headers** in some C++ environments.²³

Perf

Unlike `#include` directives, the order of `imports` is irrelevant. So, for example²⁴

```
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.²⁵ You'll see in Section 16.6 that the modules you define can specify which declarations they **export** (that is, make available) for use in other translation units, giving you **precise control over each module's interface**. Unlike an `#include`, an `import` statement does not add source code to the importing translation unit. Also, preprocessor directives in a translation unit that appear before you `import` a header unit do not affect the header unit's contents.²⁶

Import All Headers as Header Units (If Possible)

You should generally **import** all headers as header units.²⁷ Unfortunately, not all headers can be imported. You'll typically use `#include` if importing a header as a header unit produces compilation errors. This could happen if a header depends on **#defined preprocessor macros**—referred to as **preprocessor state**.

AS

Err

Compiling with Header Units in Microsoft Visual Studio

To compile any of this chapter's examples that use **header units**, ensure that your Visual Studio project is configured correctly:

1. Right-click the project name in **Solution Explorer** and select **Properties**.
2. In the **Property Pages** dialog, select **All Configurations** from the **Configuration** drop-down.
3. In the left column under **Configuration Properties > C/C++ > Language**, set the **C++ Language Standard** option to **ISO C++20 Standard (/std:c++20)** or **Preview - Features from the Latest C++ Working Draft (/std:c++latest)**.

-
22. Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer.”
 23. Cameron DaCamara, “Practical C++20 Modules and the Future of Tooling Around C++ Modules,” May 4, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
 24. Stroustrup, “Thriving in a Crowded and Changing World.”
 25. “C++20 Standard: 10 Modules—10.3 Import Declaration.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.import>.
 26. “Understanding C++ Modules: Part 3: Linkage and Fragments,” October 7, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/10/07/modules-3.html>.
 27. Daniela Engert, “Modules: The Beginner's Guide,” May 2, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

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.^{28,29,30} The following command compiles the `<iostream>` standard library header as a header unit:

```
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.³¹

Next, compile the source-code file `fig16_01.cpp` using the following command:

```
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++

In clang++, you must first precompile each header you'll import as a header unit.³² This creates a precompiled module (.pcm) file, which is specific to the clang++ compiler. The following command precompiles the `<iostream>` standard library header so we can use it as a header unit:³³

```
clang++ -std=c++20 -xc++-system-header --precompile iostream
-o iostream.pcm
```

- The `-xc++-system-header` compiler flag indicates that we are compiling a C++ standard library header.³⁴
- The `--precompile` compiler flag tells the compiler to create a precompiled module file.

28. This might change once C++20 modules are fully implemented in g++.

29. "3.23 C++ Modules." Accessed April 18, 2023. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

30. Nathan Sidwell, "C++ Modules: A Brief Tour," October 19, 2020. Accessed April 18, 2023. <https://accu.org/journals/overload/28/159/sidwell/>.

31. 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.

32. Depending on the clang++ version you use, the command `clang++` might be `clang++-16`—the version number (16) may need to be replaced with the version you've installed.

33. At the time of this writing, the compiler has a bug that produces a warning. This is a known issue.

34. There is also a `-xc++-user-header` flag for precompiling other headers as header units. See <https://clang.llvm.org/docs/StandardCPlusPlusModules.html>.

Next, compile the source-code file `fig16_01.cpp` using the following command:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_01.cpp  
-o fig16_01
```

This produces an executable named `fig16_01`, which you can execute with the command:

```
./fig16_01
```

16.5 Modules Can Reduce Translation Unit Sizes and Compilation Times

As we said, modules can reduce compilation times by eliminating repeated preprocessing of the same header files across many translation units in the same program. Consider the following simple program, which has fewer than 90 characters, including the vertical spacing and indentation:

```
#include <iostream>  
  
int main() {  
    std::cout << "Welcome to C++20 Modules!\n";  
}
```

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 processes.

When using 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 significantly reduce compilation times.



References

For more information on translation unit sizes and compilation performance, see the following resources:

- Gabriel Dos Reis and Cameron DaCamara, “Implementing C++ Modules: Lessons Learned, Lessons Abandoned,” December 18, 2021. Accessed April 18, 2023.
<https://www.youtube.com/watch?v=90WGgkuyFV8>.

- Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed April 18, 2023. <https://www.stroustrup.com/hop120main-p5-p-bfc9cd4--final.pdf>.
- Cameron DaCamara, “Practical C++20 Modules and the Future of Tooling Around C++ Modules,” May 4, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
- Rainer Grimm, “C++20: The Advantages of Modules,” May 10, 2020. Accessed April 18, 2023. <https://www.modernescpp.com/index.php/cpp20-modules>.
- Rene Rivera, “Are Modules Fast? (Revision 1),” March 6, 2019. Accessed April 18, 2023. <https://wg21.link/p1441r1>.

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 April 18, 2023. <https://docs.microsoft.com/en-us/cpp/build-insights/get-started-with-cpp-build-insights>.
- “Clang 13 Documentation: Target-Independent Compilation Options `-ftime-report` and `-ftime-trace`,” Accessed April 18, 2023. <https://clang.llvm.org/docs/ClangCommandLineReference.html#target-independent-compilation-options>.
- “Profiling the C++ Compilation Process.” Accessed April 18, 2023. <https://stackoverflow.com/questions/13559818/profiling-the-c-compilation-process>.

16.6 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:³⁵

- **export** its declaration directly,
- **export** a group of declarations enclosed in braces (`{` and `}`),
- **export** a **namespace**, which may contain many declarations, and
- **export** a **namespace member**, which also exports the **namespace’s name**³⁶ but not the **namespace’s other members**.

Recall that if an identifier’s first occurrence is its definition, it also serves as the identifier’s declaration. So each item in the preceding list may export declarations or definitions.

35. “C++ Standard: 10 Modules—10.2 Export Declaration.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.interface>.

36. So you qualify its exported members with “`name::`”.

16.6.1 module Declaration for a Module Interface Unit

Figure 16.2 defines our first **module unit**³⁷—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.

```

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 // exporting all items in the braces that follow export
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 }
```

Fig. 16.2 | Primary module interface for a module named `welcome`.

Module Declaration and Module Naming

Line 3's **module declaration** names the module `welcome`. **Module names** are lowercase identifiers (by convention) separated by dots (.)³⁸—such as `deitel.time` or `deitel.math`, which we'll use in subsequent examples. The dots do not have special meaning—in fact,

37. “C++ Standard: 10 Modules—10.1 Module Units and Purviews.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.unit>.

38. Corentin Jabot, “Naming Guidelines for Modules,” June 16, 2019. Accessed April 18, 2023. <https://wg21.link/P1634R0>.

there's a proposal to remove dot-separated naming from modules.³⁹ 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.⁴⁰ We show multi-file modules in subsequent sections.



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 16.9 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 filename (we used the name `welcome.ixx`), 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**. At the end of this section, 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.

Other Filename Extensions

Some 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**.⁴¹
- **.cpp**—Filename extension for the C++ source code in a translation unit, including module units.
- **.cppm**—A recommended **clang++** filename extension for **module units**. Visual C++ also recognizes this extension.
- **.pcm**—When you compile a **primary module interface unit** in **clang++**, the resulting file uses this extension.

39. Michael Spencer, "P1873R1: remove.dots.in.module.names," September 17, 2019. <https://wg21.link/p1873r1>.

40. "C++ Standard: 10 Modules—10.1 Module Units and Purviews."

41. DaCamara, "Practical C++20 Modules and the Future of Tooling Around C++ Modules."

16.6.2 Exporting a Declaration

You must **export a declaration** to make it available 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 16.6.5). The declarations in **export** statements must not have **internal linkage**,⁴² which includes

- 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**.”⁴³

Also, if a module defines any preprocessor macros, they're for use only in that module and cannot be exported.⁴⁴

Defining Templates, `constexpr` Functions and `inline` Functions in Modules
Similar to headers, when you **define a template, `constexpr` function or `inline` function in a module**, you must **export** its complete definition so the compiler can access it wherever the module is imported.

16.6.3 Exporting a Group of Declarations

Rather than applying **export** to individual declarations, you can **export** a group of declarations in braces (lines 13–17). Every declaration in the braces is exported. The braces do not define a new scope, so identifiers declared in such a block continue to exist beyond the block's closing brace.



16.6.4 Exporting a namespace

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will collide with an identically named variable in a different scope, possibly creating a naming conflict and resulting in errors. C++ solves this problem with **namespaces**. 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.



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 namespaces. 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

42. “C++ Standard: 10 Modules—10.2 Export Declaration.”

43. “Namespaces—Unnamed Namespaces.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/namespace#Unnamed_namespaces.

44. Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer.”

```
namespace std {
    // standard library header's declarations
}
```



indicating that the header's declarations are in namespace std. When you place **export** before a given **namespace** block, all the members in that block are **exported**, but not those in separate namespace blocks of the same namespace.

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

```
TestNamespace1::welcomeFromTestNamespace1()
```

Alternatively, you can provide a **using declaration** or **using directive** before the member is used in the translation unit. A using declaration like

```
using TestNamespace1::welcomeFromTestNamespace1;
```

brings one identifier (welcomeFromTestNamespace1) into the scope where the directive appears. A using directive like

```
using namespace TestNamespace1;
```

brings all the identifiers 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.

16.6.5 Exporting a namespace Member



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 is also exported. This does not implicitly export the namespace's other members.

16.6.6 Importing a Module to Use Its Exported Declarations



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 9.7.3). 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 were able to call each of the welcome module's exported functions.

```
1 // fig16_03.cpp
2 // Importing a module and using its exported items.
3 import <iostream>;
4 import welcome; // import the welcome module
5
```

Fig. 16.3 | Importing a module and using its exported items. (Part I of 2.)

```
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
```

Fig. 16.3 | Importing a module and using its exported items. (Part 2 of 2.)

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:

```
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..hxx`:

```
g++ -fmodules-ts -c -x c++ welcome..hxx
```

3. Compile the `main` application and link it with `welcome.o`—this command produces the **executable file** `fig16_03`:

```
g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03
```

In Step 2:

- the `-c` option says to compile `welcome..hxx`, but not link it, and
- the `-x c++` option indicates that `welcome..hxx` is a C++ file.

The `-x c++` is required because `..hxx` is not a standard `g++` filename extension. If we name `welcome..hxx` 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 `<string>` and `<iostream>` headers as **header units** because they're used in our module and our `main` application, respectively:

```
clang++ -std=c++20 -xc++-system-header --precompile string
-o string.pcm
clang++ -std=c++20 -xc++-system-header --precompile iostream
-o iostream.pcm
```

2. Compile the **module interface unit** into a precompiled module (**.pcm**) file, which is specific to the `clang++` compiler:

```
clang++ -std=c++20 -fmodule-file=string.pcm
-x c++-module welcome..hxx --precompile -o welcome.pcm
```

3. Compile the **main application** and link it with `welcome.pcm`—this command produces the **executable file** `fig16_03`:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_03.cpp
-fprebuilt-module-path=. welcome.pcm -o fig16_03
```

In Step 2:

- `-fmodule-file` specifies that our `welcome..hxx` module depends on the header unit `string.pcm`, and
- `-x c++-module` `welcome..hxx` specifies that `welcome..hxx` represents a C++ module.

The `-x c++-module` option is required here because `..hxx` is not a standard `clang++` file-name extension. If we name the file `welcome.cppm`, then `-x c++-module` is not required. In Step 3, `-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.

16.6.7 Example: Attempting to Access Non-Exported Module Contents



Modules do not implicitly export declarations—this is known as **strong encapsulation**. Thus, 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 the namespace `deitel::math`,⁴⁵ which is not exported. 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 soon see in Fig. 16.5. This is a key difference from headers—everything declared in a header can be used wherever you `#include` it.⁴⁶

```

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 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}
```

Fig. 16.4 | Primary module interface for a module named `deitel.math`. (Part 1 of 2.)

45. The namespace `deitel::math` defines a `deitel` namespace with a nested namespace `math`. The notation used here was introduced in C++17. We discuss nested namespaces in Chapter 20.

46. Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer.”

```

11 // non-exported function cube is not implicitly exported
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 2 of 2.)

It's good practice in a module to **put exported identifiers in namespaces** to help avoid  SE name collisions when multiple modules export the same identifier. We found in our research that **namespace names** typically mimic their **module names**⁴⁷—so for the `deitel.math` module, we specified the namespace `deitel::math` (line 5).

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 `square` was exported by the `deitel.math` module. Line 12, however, results in compilation errors—the `deitel.math` module did not export `cube`. The compilation errors in Fig. 16.5 are from Visual C++. We highlighted key error messages in bold and added vertical spacing for readability.



```

1 // fig16_05.cpp
2 // Showing that a module's non-exported identifiers are inaccessible.
3 import <iostream>;
4 import deitel.math; // import the deitel.math 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 }

```

Fig. 16.5 | Showing that a module's non-exported identifiers are inaccessible. (Part 1 of 2.)

47. Daniela Engert, “Modules: The Beginner’s Guide,” May 2, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.

```

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

```

Fig. 16.5 | Showing that a module's non-exported identifiers are inaccessible. (Part 2 of 2.)

g++ Error Messages

Err  To see the g++ error messages, execute the following commands, each of which we explained in Section 16.6.6:

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:

```

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';
      |           ^
      |           ~~~

```

clang++ Error Messages

Err  To see the clang++ error messages, execute the following commands, each of which we explained in Section 16.6.6:

1. clang++ -std=c++20 -xc++-system-header --precompile iostream
-o iostream.pcm
2. clang++ -std=c++20 -x c++-module deitel.math.ixx --precompile
-o deitel.math.pcm
3. clang++ -std=c++20 -fmodule-file=deitel.math.pcm
-fmodule-file=iostream.pcm fig16_05.cpp -o fig16_05

We highlighted the key error message in bold:

```
fig16_05.cpp:12:47: error: declaration of 'cube' must be imported from module 'deitel.math' before it is required
    std::cout << "cube(3) = " << deitel::math::cube(3) << '\n';
                                         ^
/usr/src/lesson16/fig16_04-05/deitel.math.ixx:12:8: note: declaration here is
not visible
    int cube(int x) {
        ^
1 error generated.
```

16.7 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**⁴⁸

```
module;
```

which you place before the `module` declaration at the beginning of a module unit. The **global module fragment** may contain **only preprocessor directives**. A module interface unit can export a declaration that was `#included` in the global module fragment, so other implementation units that `import` the module can use that declaration.⁴⁹ **Global module fragments** from all **module units** are placed into the **global module**, which also contains non-modularized code in **non-module translation units**, such as the one containing `main`.



16.8 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. Modules also support separating interface from implementation:

- You can split your interface and implementation into separate files (shown in Sections 16.8.1–16.8.2).
- You can define your interface and implementation in one source file (discussed in Section 16.8.3).

16.8.1 Example: Module Implementation Units

Sometimes it's helpful to break a large 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 use a **primary module interface unit** for a module's interface and a separate source code file for the module's implementation details.



48. “C++ Standard: 10 Modules—10.4 Global Module Fragment.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.global.frag>.

49. Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer.”

Primary Module Interface Unit

The `deitel.math` module's **primary module interface unit** (Fig. 16.6) exports the `deitel::math` namespace (lines 7–10) containing a function prototype for the function `average`, which calculates the average of a `vector<int>`'s elements.

```

1 // Fig. 16.6: 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 };

```

Fig. 16.6 | 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. 16.7) are **module implementation units** (typically defined in .cpp files).⁵⁰ These can **split larger modules into multiple source files** to make the code more manageable. Line 3 indicates that this file is a module implementation unit for the `deitel.math` module. A **module implementation unit implicitly imports its module's interface**. The compiler combines the primary module interface unit and its corresponding module implementation unit(s) into a single **named module** that other translation units can `import`.⁵¹ Lines 10–13 implement the `average` function in a namespace, which must have the same name as the one containing `average`'s prototype in the primary module interface unit (Fig. 16.6).

```

1 // Fig. 16.7: 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 };

```

Fig. 16.7 | Module implementation unit for the module `deitel.math`.

50. "C++20 Standard: 10 Modules—10.1 Module Units and Purviews." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module#unit>.

51. "C++20 Standard: 10 Modules." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module>.

Using the Module

The `main` program in Fig. 16.8 imports module `deitel.math` (line 7) and calls its `average` function (line 17) to calculate the average of the `integers` vector's elements.

```
1 // fig16_08.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
```

Fig. 16.8 | 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 16.6.1. Ensure that your project includes in its **Source Files** folder:

- `deitel.math.ixx`—the **primary module interface unit**,
- `deitel.math-impl.cpp`⁵²—the **module implementation unit**, and
- `fig16_08.cpp`—the **main application**.

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 16.6.6 to compile the standard library headers `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as header units. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Then, compile the module implementation unit:

```
g++ -fmodules-ts -c deitel.math-impl.cpp
```

52. We named the module implementation unit with "-impl" in the filename to support the compilation steps of the `g++` compiler, ensuring that each resulting `.o` file will have a unique name.

We now have the object files `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`:

```
g++ -fmodules-ts fig16_08.cpp deitel.math.o deitel.math-impl.o
-o fig16_08
```

Compiling This Example in clang++

In clang++, use the commands from Section 16.6.6 to compile the standard library headers `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as header units. Next, compile the primary module interface unit into a precompiled module (.pcm) file:

```
clang++ -std=c++20 -fmodule-file=vector.pcm
-x c++-module deitel.math.ixx --precompile -o deitel.math.pcm
```

Next, compile the module implementation unit into an object file:

```
clang++ -std=c++20 -fmodule-file=deitel.math.pcm
-fmodule-file=vector.pcm -fmodule-file=numeric.pcm
-c deitel.math-impl.cpp -o deitel.math-impl.o
```

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 the files `deitel.math-impl.o` and `deitel.math.pcm`:

```
clang++ -std=c++20 -fmodule-file=algorithm.pcm
-fmodule-file=iostream.pcm -fmodule-file=iterator.pcm
-fmodule-file=vector.pcm fig16_08.cpp deitel.math-impl.o
-fprebuilt-module-path=. deitel.math.pcm -o fig16_08
```

16.8.2 Example: Modularizing a Class

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. 16.9) defines and exports the namespace `deitel::time` (lines 7–19) containing the `Time` class definition (lines 8–18).

```

1 // Fig. 16.9: deitel.time.ixx
2 // Primary module interface for a simple Time class.
3 export module deitel.time; // declare the 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;
```

Fig. 16.9 | Primary module interface for a simple `Time` class. (Part I of 2.)

```

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 }

```

Fig. 16.9 | Primary module interface for a simple Time class. (Part 2 of 2.)

deitel.time Module Implementation Unit

Line 4 of Fig. 16.10 indicates that `deitel.time-impl.cpp` is a **deitel.time module implementation unit**. The rest of the file defines class `Time`'s member functions. Line 8 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.

```

1 // Fig. 16.10: 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 }

```

Fig. 16.10 | `deitel.time` module implementation unit containing the `Time` class member function definitions.

Using Class Time from the `deitel.time` Module

The program in Fig. 16.11 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

1 // fig16_11.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 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
```

Fig. 16.11 | Importing the `deitel.time` module and using its `Time` class.

Compiling This Example in Visual C++

Add `deitel.time.ixx` to your Visual C++ project, as in Section 16.6.1. Next, ensure that your project includes in its **Source Files** folder:

- `deitel.time.ixx`—the primary module interface unit,
- `deitel.time-impl.cpp`—the module implementation unit, and
- `fig16_11.cpp`—the main application file.

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 16.6.6 to compile the standard library headers `<iostream>`, `<string>` and `<stdexcept>` as header units.

Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.time.ixx
```

Then, compile the module implementation unit:

```
g++ -fmodules-ts -c deitel.time-impl.cpp
```

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`:⁵³

```
g++ -fmodules-ts fig16_11.cpp deitel.time.o deitel.time-impl.o  
-o fig16_11
```

Compiling This Example in clang++

In clang++, use the commands introduced in Section 16.6.6 to compile the standard library headers `<iostream>`, `<string>` and `<stdexcept>` as header units.

Next, compile the primary module interface unit into a precompiled module (.pcm) file:

```
clang++ -std=c++20 -fmodule-file=string.pcm  
-x c++-module deitel.time.ixx --precompile -o deitel.time.pcm
```

Next, compile the module implementation unit into an object file:

```
clang++ -std=c++20 -fmodule-file=deitel.time.pcm  
-fmodule-file=string.pcm -fmodule-file=stdexcept.pcm  
-c deitel.time-impl.cpp -o deitel.time-impl.o
```

The option `-fmodule-file=deitel.time.pcm` specifies the name of the module's primary module interface unit. Finally, compile the main application and link it with `deitel.time-impl.o` and `deitel.time.pcm`:

```
clang++ -std=c++20 fig16_11.cpp deitel.time-impl.o  
-fmodule-file=iostream.pcm -fmodule-file=string.pcm  
-fmodule-file=stdexcept.pcm -fprebuilt-module-path=.  
deitel.time.pcm -o fig16_11
```

53. At the time of this writing, this example is not linking correctly in g++.

16.8.3 :private Module Fragment

Modules also support separating interface from implementation in one translation unit by using a **:private module fragment** in the primary module interface unit, as in:^{54,55,56}

```
export module name; // introduces the module name
// code that defines the primary module interface
// private module fragment for defining implementation details
module :private;
// implementation details
```

 With this approach to defining a module, the primary module interface unit must be the module's only module unit. Changes to the implementation details in the **:private module fragment** do not affect this module's interface, nor do they affect other translation units that **import** this module.⁵⁷

 Use the **:private module fragment** when you want to manage your interface and implementation details in one translation unit. The **:private module fragment** is essentially a **module implementation unit** after

```
module :private;
```

 but you don't need to compile a separate .cpp file. Another potential benefit is that having all the code in a single interface may enable the compiler to better optimize the code."⁵⁸

16.9 Partitions

You can divide a module's interface and/or its implementation into smaller pieces called **partitions**. 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.⁵⁹ Whether items are recompiled would be determined by the compiler. The compiler aggregates a module's partitions into a single named module for import into other translation units.

54. "C++20 Standard: 10 Modules—Private module fragment." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.private.frag>.

55. Cameron DaCamara, "Standard C++20 Modules Support with MSVC in Visual Studio 2019 Version 16.8—Private Module Fragments," September 14, 2020, Accessed April 18, 2023. <https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/#private-module-fragments>.

56. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

57. "C++20 Standard: 10 Modules—Private Module Fragment." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.private.frag>.

58. This paragraph's contents are based on a February 10, 2022 e-mail interaction between Paul Deitel and Cameron DaCamara, Senior Software Engineer at Microsoft and part of the Visual C++ team.

59. DaCamara, "Practical C++20 Modules and the Future of Tooling Around C++ Modules."

16.9.1 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. 16.12 indicates that the translation unit `deitel.math-powers..hxx` 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.⁶⁰



```

1 // Fig. 16.12: 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. 16.12 | Module interface partition unit `deitel.math:powers`.

`deitel.math:roots` Module Interface Partition Unit

Line 3 of Fig. 16.13 indicates that the translation unit `deitel.math-roots..hxx` is a **module interface partition unit** with the **partition name** "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.⁶¹
- 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.



60. "Modules—Module Partitions." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/modules>.

61. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

```

1 // Fig. 16.13: 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. 16.13 | Module interface partition unit deitel.math:roots.**deitel.math Primary Module Interface Unit**

Figure 16.14 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 **export import** the module interface partition units:

- First, each **import** imports the **module interface partition unit** that follows the colon (:)—in this case, `powers` or `roots`.
- Next, the **export** keyword before each **import** indicates that the specified **module interface partition unit**'s exported members should be part of the `deitel.math` module's **primary module interface**.



The users of your module cannot see its partitions.⁶²

```

1 // Fig. 16.14: 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;
```

Fig. 16.14 | 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.

62. "C++ Standard: 10 Modules—10.1 Module Units and Purviews." Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.unit>.

Using the `deitel.math` Module

Figure 16.15 imports the `deitel.math` module (line 4) and lines 9–12 use its exported functions to demonstrate that this primary module interface contains all the functions exported by the `powers` and `roots` module interface partitions.

```

1 // fig16_15.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

```

Fig. 16.15 | 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 16.6.1, then add the file `fig16_15.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 16.6.6 to compile the standard library headers `<cmath>` and `<iostream>` as header units. Next, compile each module interface partition unit:

```

g++ -fmodules-ts -c -x c++ deitel.math-powers.ixx
g++ -fmodules-ts -c -x c++ deitel.math-roots.ixx

```

Then, compile the primary module interface unit:

```

g++ -fmodules-ts -c -x c++ deitel.math.ixx

```

Finally, compile the **main application** and link it with `deitel.math-powers.o`, `deitel.math-roots.o` and `deitel.math.o`:

```

g++ -fmodules-ts fig16_15.cpp deitel.math-powers.o
deitel.math-roots.o deitel.math.o -o fig16_15

```

Compiling This Example in clang++

When building a module with partitions, you must build the partitions before the primary module interface unit. At the time of this writing, clang++ would not compile the `<cmath>` header into a header unit. So before compiling this example in clang++, change the `import` statement in line 5 of Fig. 16.13 to

```
#include <cmath>
```

In clang++, use the commands introduced in Section 16.6.6 to compile the standard library header `<iostream>` as a header unit. Next, compile each module interface partition unit into a precompiled module (.pcm) file:

```
clang++ -std=c++20 -x c++-module deitel.math-powers..hxx
--precompile -o deitel.math-powers.pcm
clang++ -std=c++20 -x c++-module deitel.math-roots..hxx
--precompile -o deitel.math-roots.pcm
```

Then, compile the primary module interface unit into a precompiled module (.pcm) file:

```
clang++ -std=c++20 -x c++-module deitel.math..hxx
-fprebuilt-module-path=. --precompile -o deitel.math.pcm
```

Finally, compile the main application and link it with `deitel.math-powers.pcm`, `deitel.math-roots.pcm` and `deitel.math.pcm`:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_15.cpp
-fprebuilt-module-path=. deitel.math.pcm deitel.math-powers.pcm
deitel.math-roots.pcm -o fig16_15
```

16.9.2 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**.⁶³



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 the `export` keyword:

```
module ModuleName:PartitionName;
```

Module implementation partition units do not implicitly import the primary module interface.⁶⁴ At the time of this writing, none of our preferred compilers support module implementation partitions, so we do not show them here.

16.9.3 Example: “Submodules” vs. Partitions



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

63. Richard Smith, “Merging Modules—Section 2.2 Module Partitions,” February 22, 2019. Accessed April 18, 2023. <https://wg21.link/p1103r3>.

64. C++ Standard, “10 Modules—10.1 Module Units and Purviews.”

their interfaces. Let's reimplement Section 16.9's `deitel.math` module using only **primary module interface units** to demonstrate the flexibility this provides to developers.

`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. 16.16). Rather than declaring a **module interface partition**, as in Fig. 16.12

```
export module deitel.math:powers;
```

line 3 of Fig. 16.16 declares a **primary module interface unit** with a dot-separated name:

```
export module deitel.math.powers;
```

We can now independently import `deitel.math.powers` and use its functions (Fig. 16.17).

```
1 // Fig. 16.16: 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 }
```

Fig. 16.16 | Primary module interface unit `deitel.math.powers`.

```
1 // fig16_17.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
```

Fig. 16.17 | Using the `deitel.math.powers` module's functions.

Compiling This Example in Visual C++

Add the `deitel.math.powers.ixx` and `fig16_17.cpp` files to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in g++

Use the commands introduced in Section 16.6.6 to compile the standard library header `<iostream>` as a header unit. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.powers.ixx
```

Then, compile the main application and link it with `deitel.math.powers.o`:

```
g++ -fmodules-ts fig16_17.cpp deitel.math.powers.o -o fig16_17
```

Compiling This Example in clang++

Use the commands introduced in Section 16.6.6 to compile the standard library header `<iostream>` as a header unit. Next, compile the primary module interface unit into a pre-compiled module (.pcm) file:

```
clang++ -std=c++20 -x c++-module deitel.math.powers.ixx
--precompile -o deitel.math.powers.pcm
```

Finally, compile the main application and link it with `deitel.math.powers.pcm`:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_17.cpp
-fprebuilt-module-path=. deitel.math.powers.pcm -o fig16_17
```

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.roots` a **primary module interface unit** (Fig. 16.18). Rather than declaring a **module interface partition**, as in Fig. 16.13

```
export module deitel.math:roots;
```

line 3 of Fig. 16.18 declares a **primary module interface unit** with a dot-separated name:

```
export module deitel.math.roots;
```

We can now independently import `deitel.math.roots` and use its functions (Fig. 16.19).

```

1 // Fig. 16.18: 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. 16.18 | Primary module interface unit `deitel.math.roots`.

```

1 // fig16_19.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
```

Fig. 16.19 | Using the `deitel.math.roots` module's functions. (Part 1 of 2.)

```

6  using namespace deitel::math;
7
8  int main() {
9      std::cout << "squareRoot(9): " << squareRoot(9)
10     << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
11 }

```

```

squareRoot(9): 3
cubeRoot(1000): 10

```

Fig. 16.19 | Using the `deitel.math.roots` module's functions. (Part 2 of 2.)

Compiling This Example in Visual C++

Add the `deitel.math.roots.ixx` and `fig16_19.cpp` files to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in g++

Use the commands introduced in Section 16.6.6 to compile the standard library headers `<iostream>` and `<cmath>` as header units. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.roots.ixx
```

Then, compile the main application and link it with `deitel.math.roots.o`:

```
g++ -fmodules-ts fig16_19.cpp deitel.math.roots.o -o fig16_19
```

Compiling This Example in clang++

At the time of this writing, `clang++` would not compile the `<cmath>` header into a header unit. In a primary module interface, `#include` directives should be placed in the global module fragment. So before compiling this example in `clang++`, remove the `import` statement in line 5 of Fig. 16.18 and place the following lines before the module declaration:

```
module;
#include <cmath>
```

Use the commands introduced in Section 16.6.6 to compile the standard library header `<iostream>` as a header unit. Next, compile the primary module interface unit into a pre-compiled module (`.pcm`) file:

```
clang++ -std=c++20 -x c++-module deitel.math.roots.ixx
--precompile -o deitel.math.roots.pcm
```

Finally, compile the main application and link it with `deitel.math.powers.pcm`:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_19.cpp
-fprebuilt-module-path=. deitel.math.roots.pcm -o fig16_19
```

deitel.math Primary Module Interface Unit

Figures 16.16 and 16.18 are now separate modules. The names `deitel.math.powers` and `deitel.math.roots` imply a logical relationship between them, and both modules export the `deitel::math` namespace, but they do not define one module. For convenience, we can aggregate these separate modules in a primary module interface unit that **exports imports** both “submodules,” as shown in lines 7 and 8 of Fig. 16.20. We can then use all the functions from both “submodules” by importing `deitel.math` (Fig. 16.21).





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.

```

1 // Fig. 16.20: 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;
```

Fig. 16.20 | Primary module interface unit `deitel.math` aggregates declarations from “submodules” `deitel.math.powers` and `deitel.math.roots`.

```

1 // fig16_21.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
```

Fig. 16.21 | Using the `deitel.math` module's functions.

Compiling This Example in Visual C++

Add the Fig. 16.16, 16.18 and 16.20 .ixx files and `fig16_21.cpp` to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in g++

For these steps, we assume you're executing commands in the same folder where you compiled `deitel.math.powers.ixx` and `deitel.math.roots.ixx`. Compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Then, compile the main application and link it with `deitel.math.powers.o`, `deitel.math.roots.o` and `deitel.math.o`:

```
g++ -fmodules-ts fig16_21.cpp deitel.math.powers.o  
deitel.math.roots.o deitel.math.o -o fig16_21
```

Compiling This Example in clang++

For these steps, we assume you're executing commands in the same folder where you compiled `deitel.math.powers.ixx` and `deitel.math.roots.ixx`. Compile the primary module interface unit into a precompiled module (.pcm) file:

```
clang++ -std=c++20 -x c++-module deitel.math.ixx  
-fprebuilt-module-path=. --precompile -o deitel.math.pcm
```

Then, compile the main application and link it with `deitel.math.powers.pcm`, `deitel.math.roots.pcm` and `deitel.math.pcm`:

```
clang++ -std=c++20 -fmodule-file=iostream.pcm fig16_21.cpp  
-fprebuilt-module-path=. deitel.math.pcm deitel.math.powers.pcm  
deitel.math.roots.pcm -o fig16_21
```

16.10 Additional Modules Examples

The next several subsections demonstrate additional modules concepts:

- 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.

16.10.1 Example: Importing the C++ Standard Library as Modules

C++23 requires a **modularized standard library** containing two modules named `std` and `std.compat`.⁶⁵ Most programs will import the standard library as follows:

```
import std;
```

At the time of this writing, only clang++ versions 15 and 16 support this `import` with their early access modularized C++ standard library.

Microsoft Visual C++ has an experimental modularized C++ standard library containing several modules. You can `import` the following modules into your Visual C++ projects and “potentially speed up compilation times depending on the size of your project”:⁶⁶

- `std.core`—This module contains most of the standard library except for the following items.
- `std.filesystem`—Module containing the `<filesystem>` header’s capabilities.
- `std.memory`—Module containing the `<memory>` header’s capabilities.



65. Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup and Jonathan Wakely, “Standard Library Modules `std` and `std.compat`,” October 13, 2021. Accessed April 18, 2023. <https://wg21.link/p2465>.

66. Colin Robertson and Nick Schonning, “Overview of Modules in C++,” December 13, 2019. Accessed April 18, 2023. <https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160>.

- `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>`.

Figure 16.22 imports the `std.core` module (line 3), then uses `cout` from the standard library’s `<iostream>` capabilities to output a string.

```

1 // fig16_22.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!

Fig. 16.22 | 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, search for “C++ Modules” and ensure that **C++ Modules for v### build tools** is checked—at the time of this writing, **###** is **143**. 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 **ISO C++20 Standard (/std:c++20)**.

You can now compile and run Fig. 16.22.

Modifying and Compiling the Program in clang++

To compile this program in clang++, change line 3 of Fig. 16.22 to

```
import std;
```

Next, compile the program using the following command:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
-fimplicit-module-maps fig16_22.cpp -o fig16_22
```

16.10.2 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.^{67,68} Figures 16.23 and 16.24 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. 16.23 and line 5 of Fig. 16.24—`moduleA` imports `moduleB`, which in turn imports `moduleA`.

```
1 // Fig. 16.23: 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
```

Fig. 16.23 | Primary module interface unit that imports moduleB.

```
1 // Fig. 16.24: 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
```

Fig. 16.24 | Primary module interface unit that imports moduleA.

Compiling Figs. 16.23 and 16.24 in Visual C++ results in the following error message: 

```
error : Cannot build the following source files because there is a
cyclic dependency between them: ch16\fig16_23-24\moduleA.ixx depends
on ch16\fig16_23-24\moduleB.ixx depends on ch16\fig16_23-24\
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.

67. “C++ Standard: 10 Modules—10.3 Import Declaration.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.import>.

68. “Understanding C++ Modules: Part 2: `export`, `import`, `Visible`, and `Reachable`,” March 31, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/03/31/modules-2.html>.

16.10.3 Example: imports Are Not Transitive



In Section 16.6.7, 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. 16.25) and **moduleB** (Fig. 16.26)—**moduleB** imports but does not **re-export moduleA** (line 6 of Fig. 16.26). As a result, **moduleA**’s exported **cube** function is not part of **moduleB**’s interface.

```

1 // Fig. 16.25: moduleA..hxx
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; }
```

Fig. 16.25 | Primary module interface unit that exports function **cube**.

```

1 // Fig. 16.26: moduleB..hxx
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; }
```

Fig. 16.26 | Primary module interface unit that imports, but does not export, **moduleA** and exports function **square**.

Figure 16.27 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

```

1 // fig16_27.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 }
```

Fig. 16.27 | Showing that **moduleB** does not implicitly export **moduleA**’s function.

16.10.4 Example: Visibility vs. Reachability

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.



Some declarations are **reachable** but not **visible**, meaning you cannot explicitly mention the declaration's name in another translation unit, but the declaration is indirectly accessible.^{69,70,71} Anything visible is reachable, but not vice versa. The easiest way to understand this concept is with code. To demonstrate **reachability**, we modified Fig. 16.9's primary module interface unit `deitel.time.ixx`.⁷² There are two key changes (Fig. 16.28):



- 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**.

The module's implementation unit (Fig. 16.10) is identical for this example, so we do not show it here.

```

1 // Fig. 16.28: 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    }

```

Fig. 16.28 | Primary module interface unit for the `deitel.time` module.

69. “C++ Standard: 10 Modules—10.7 Reachability.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module.reach>.

70. “Understanding C++ Modules: Part 2: `export`, `import`, `Visible`, and `Reachable`.”

71. Richard Smith, “Merging Modules—Section 2.2 Module Partitions,” February 22, 2019. Accessed April 18, 2023. <https://wg21.link/p1103r3>.

72. At the time of this writing, this example does not link correctly in `g++` and does not compile in `clang++`.

The program in Fig. 16.29 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 `deitel::time::Time`, you'd get an error like the following (from Visual C++):

```
'Time': is not a member of 'deitel::time'
```

This error occurs because `Time` is not visible in this translation unit. However, `Time`'s definition is reachable because `getTime` returns a `Time` object—the compiler knows this, so it

Mod

can infer the variable `t`'s type. When a class definition is reachable, the class'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 16.8.2, except that the main program's filename is now `fig16_29.cpp`.

```

1 // fig16_29.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
```

Fig. 16.29 | Showing that type `deitel::time::Time` is reachable and its public members are visible.

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 our favorites are listed here. 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 April 18, 2023. <https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/>.
- Steve Downey, “Writing a C++20 Module,” July 5, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=A04piAqV9mg>.
- Daniela Engert, “Modules: The Beginner’s Guide,” May 2, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.
- Yuka Takahashi, Oksana Shadura and Vassil Vassilev, “Migrating Large Codebases to C++ Modules,” August 22, 2019. Accessed April 18, 2023. <https://arxiv.org/abs/1906.05092>.
- Nathan Sidwell, “Converting to C++20 Modules,” October 4, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=KVsWIEw3TTw>.

We will post additional resources as they become available at

<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**.⁷³ C++20 modules are so new that the tooling to help you use modules 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.

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*.⁷⁴
- Python has a well-developed module system, which we used extensively in our Python books.^{75,76}
- Microsoft’s .NET platform languages like C# and Visual Basic can modularize code using assemblies.

Wikipedia lists several dozen languages with modules capabilities.⁷⁷

73. “To Boldly Suggest an Overall Plan for C++23,” November 25, 2019. <https://wg21.link/p0592r4>

74. 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>.

75. Paul Deitel and Harvey Deitel, *Python for Programmers*, 2019. Pearson Education, Inc.

76. Paul Deitel and Harvey Deitel, *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*, 2020. Pearson Education, Inc.

77. “Modular Programming.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Modular_programming.

Many languages provide tooling to help you work with their modules systems and modularize your code. The following tooling might eventually appear in the C++ ecosystem:

- **Module-aware build tools** that manage compiling software systems (Visual C++ already has this)
- **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 Bloomberg⁷⁸ 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 (listed reverse chronological order) discuss various C++ standard and third-party vendor opportunities for module-aware tools that will improve the C++ development process:

- **Daniel Ruoso**, “Requirements for Usage of C++ Modules at Bloomberg,” July 12, 2021. Accessed April 18, 2023. <https://isocpp.org/files/papers/P2409R0.pdf>.
- **Nathan Sidwell**, “P1184: A Module Mapper,” July 10, 2020. Accessed April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <https://www.youtube.com/watch?v=ow2zV0Udd9M>.
- **Nathan Sidwell**, “C++ Modules and Tooling,” October 4, 2018. Accessed April 18, 2023. https://www.youtube.com/watch?v=4y0Z8Zp_Zfk.
- **Gabriel Dos Reis**, “Modules Are a Tooling Opportunity,” October 16, 2017. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0822r0.pdf>.

78. Daniel Ruoso, “Requirements for Usage of C++ Modules at Bloomberg,” July 12, 2021. Accessed April 18, 2023. <https://wg21.link/P2409R0>.

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 `importing` 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 everything reachable is not necessarily visible.

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 tooling that might appear in the next several years. For your further study, the appendices following the exercises provide lists of videos, articles, technical papers and documentation that we referenced as we wrote this chapter. We also include a glossary with key modules-related 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 (<https://www.jetbrains.com/lp/devcosystem-2021/>) showed that 72% of Java developers are still working in some capacity with Java 8—the version of Java before JPMS was introduced. Organizations using C++ will likely proceed with caution, too. Many will wait

to learn about other organizations' experiences with modularizing large legacy codebases and launching new modularized software-development projects.

In the next chapter, we present C++ techniques that enable you to take advantage of your system's multi-core architecture—concurrency, parallelism and the parallel standard-library algorithms.

Self-Review Exercises

16.1 Define each of the following modules terms:

- a) `import`
- b) `export`
- c) `module`
- d) `export module`
- e) `module unit`
- f) `:private`
- g) `partition`

16.2 Fill in the blanks in each of the following statements:

- a) The _____ 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.
- b) A module's interface specifies the module members the module makes available for use in other translation units. You do this by _____.
- c) 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 _____, each of which defines a scope in which identifiers and variables are placed.
- d) Modules have _____—they do not implicitly export declarations, giving you precise control over the declarations you `export` for use in other translation units.
- e) All files containing `module` declarations without the `export` keyword are _____ (typically defined in .cpp files). These can split larger modules into multiple source files to make the code more manageable.
- f) Every module must have one primary module interface unit with a(n) _____ declaration that does not include a partition name.
- g) A declaration is _____ 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.

16.3 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Modules help developers be more productive, especially as they build, maintain and evolve small software systems.
- b) Because the preprocessor understands C++, it is able to perform code optimizations before the compiler executes.
- c) A header unit's declarations are available to the importing translation unit because header units implicitly "export" their contents.
- d) The global module fragment may contain only preprocessor directives.
- e) You can divide a module's interface and/or its implementation into smaller pieces called partitions. When working on large projects, this can help you or-

ganize 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.

- f) Module partitions are visible outside their module and can be imported into any translation unit.
- g) All module interface partitions with the same module name are part of the same module, are implicitly known to one another and implicitly import the module's interface.
- h) In a module implementation partition, the `module` declaration must contain the `export` keyword.
- i) A module is not allowed to have a dependency on itself—that is, a module cannot import itself directly or indirectly.
- j) When a class definition is reachable, the class's members become visible.

Answers to Self-Review Exercises

16.1 See the answers below:

- a) Makes a module's `exported` declarations available in a translation unit.
- b) Makes a declaration available to translation units that `import` the corresponding module.
- c) Every module unit has a `module` declaration specifying the module's name and possibly a partition name.
- d) Indicates that a module unit is the primary module interface unit and introduces the module's name.
- e) An implementation unit containing a `module` declaration.
- f) 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.
- g) 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.

16.2 a) linker. b) exporting the member's declaration using the `export` keyword. c) namespaces. d) strong encapsulation. e) module implementation units. f) `export module`. g) visible.

16.3 See the answers below:

- a) False. Actually, modules help developers be more productive, especially as they build, maintain and evolve large software systems.
- b) False. Actually, the preprocessor is simply a text-substitution mechanism—it does not understand C++.
- c) True.
- d) True.
- e) True.
- f) False. Actually, module partitions are not visible outside their module, so they cannot be imported into translation units that are not part of the same module.
- g) False. Actually, though all module interface partitions with the same module name are part of the same module, they are not implicitly known to one another, and they do not implicitly import the module's interface.

- h) False. Actually, in a module implementation partition, the `module` declaration must not contain the `export` keyword.
- i) True.
- j) True.

Exercises

16.4 Fill in the blanks in each of the following statements:

- a) 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 _____.
- b) Rather than `#including` a header's contents in a source-code file, you can `import` a header. 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 _____.
- c) It's good practice in a module to put exported identifiers in _____ to help avoid name collisions when multiple modules export the same identifier.
- d) Modules also support separating interface from implementation in one translation unit by using a _____ in the primary module interface unit.

16.5 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Every module has one primary module interface unit containing an `export module` declaration that introduces the module's name. The primary module interface unit specifies the module members that the client code can access.
- b) Importing a module inserts the module's code into a translation unit. So, like headers, modules need include guards.
- c) Classes can be defined using `.h` headers and `.cpp` source-code files to separate a class's interface from its implementation. Modules also support separating interface from implementation in which each must be in separate files.
- d) A module implementation unit implicitly imports the interface for the specified module name.
- e) The compiler aggregates a module's partitions into a single named module for import into other translation units.
- f) If you `export import` a header unit, its preprocessor macros are implicitly re-exported, so they are available to all translation units that import your module.
- g) When building a module with partitions, you must build the partitions before the primary module interface unit.
- h) Module implementation partition units implicitly `import` the primary module interface.
- i) An example of modules tooling is that Visual Studio enables you to add modules to your projects, and its build tools can compile and link your modularized applications.

16.6 (Modules: Benefits) Discuss how modules can offer immediate benefits in every program, even in small systems.

16.7 (Modules: Tools) Explain what each of the following categories of module-related tools does: a) module discovery tools, b) dependency-checking tools, c) tools that visualize module dependencies and d) module-aware build tools.

16.8 (Modules: Imports Are Not Transitive) Explain what it means when we say that “import statements are not transitive.”

Modules Code Exercises

16.9 (Modules: Importing Standard Library Headers as Header Units) The program in Fig. 14.1 `#includes` several standard library headers. Modify Fig. 14.1 to replace each header’s `#include` with a corresponding `import` statement. Compile and run the program to confirm that it produces the same results shown in Fig. 14.1.

16.10 (Modules: Creating a Module) The program in Fig. 5.10 defined overloaded `square` functions. Refactor that program’s code as follows:

- Create a module named `deitel.squares` containing the two `square` overloads.
- In the module, place the functions in the namespace `deitel::squares` and `export` the namespace’s contents.
- Create a separate .cpp file for the `main` function. In this file, `import` the module and modify your code to use the versions of the overloaded functions from the `deitel::squares` namespace.

Compile the program and run it to confirm that it produces the same results shown in Fig. 5.10.

16.11 (Modules: Creating a Module) Modify your module from Exercise 16.10 so that rather than exporting the entire `deitel::squares` namespace, it exports only the `square` function that receives an `int` parameter. Does the `main` application still compile? If so, when you run it, does it produce the same output as in Fig. 5.10?

16.12 (Modules: Separating Interface from Implementation) Section 9.22 defined a custom `Cipher` class that implemented the Vigenère secret-key cipher algorithm. Refactor that example:

- Create a module named `deitel.encryption` containing the `Cipher` class’s definition.
- In the module, separate the `Cipher` class’s interface from its implementation, as demonstrated in Section 16.8.2.
- Place the class in namespace `deitel::encryption` and `export` the namespace.
- In the `main` application’s .cpp file, `import` the `deitel.encryption` module, and qualify class `Cipher`’s name with the `deitel::encryption` namespace.

Compile the program and run it to confirm that it produces the same results shown in Section 9.22.

16.13 (Modules: :private Module Fragment) Modify your solution to Exercise 16.12 to define the entire class in the primary module interface unit.

16.14 (Modules: Partitions) Using the techniques shown in Section 16.9.1, modify your solution to Exercise 16.10 to define each `square` function in a separate partition.

16.15 (Modules: Reduced Translation Unit Sizes) The program in Fig. 14.1 `#includes` several standard library headers. For your compiler, use the techniques we discussed in Section 16.5 to preprocess Fig. 14.1 and determine its translation unit size in bytes. Then,

change the `#include` directives to `import` statements and preprocess the program again. Compare the sizes of the resulting translation units.

16.16 (Modules: Cyclic Dependencies Are Not Allowed) Define two modules that attempt to import one another. Show the error caused by this cyclic dependency.

16.17 (Modules: Visibility vs. Reachability) Using the `Account` class from Fig. 9.2, create a `deitel.account` module. In the module, define the namespace `deitel::account` containing the `Account` class definition. Do not export the namespace or the class. Instead, define in the namespace a function named `createAccount` that takes no arguments, creates an `Account` object and returns it by value. Then, export that function. Modify Fig. 9.1's `main` program to `import` the `deitel.account` module. Then, demonstrate that although the type `Account` is not visible in the `main` program, it is reachable. In particular, show that you can store the `Account` object returned by `createAccount` and interact with it.

Modules Code Projects

16.18 (Modules Project: Compilation Performance) Investigate the compiler profiling tools referenced in Section 16.5, then compare the performance of compiling Fig. 14.1 with `#include` directives vs. `import` statements for the standard library headers.

16.19 (Modules Project: Importing the Standard Library) In C++23, you'll be able to import the entire C++ standard library using

```
import std;
```

Research whether your compiler supports the modularized C++ standard library and if so, modify the program of Fig. 14.1 to replace the four `#include` preprocessor directives with the preceding import statement. Compile the program and run it to confirm that it produces the same results.

Modules Research Projects

16.20 (Modules Research Project: Not All Headers Can Be Imported as Header Units) Search the Internet for the typical header contents that would prevent a header from being used as a header unit.

16.21 (Modules Research Project: Guidelines) At the time of this writing, the C++ Core Guidelines at

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

have not been updated for C++20 modules. Search the Internet for the latest recommendations and guidelines on using modules in new code bases and transitioning existing code bases to modules.

16.22 (Modules Research Project: Companies Using C++20 Modules) Search the Internet for case studies on the experiences of companies using C++20 modules.

16.23 (Modules Research Project: Migrating to C++20 Modules) Search the Internet for case studies on the experiences of companies migrating existing code bases to C++20 modules.

16.24 (Modules Research Project: Modules Tooling) Search the Internet for information about your compiler's current and forthcoming tooling support for C++20 modules.

Appendix: Modules Videos Bibliography

Videos are listed in reverse chronological order.

- Bill Hoffman, “**import CMake, CMake and C++20 Modules - CppCon 2022**,” November 2, 2022. Accessed April 18, 2023. <https://www.youtube.com/watch?v=5X803cXe02Y>.
- Chuanqi Xu, “**Compilation Speedup Using C++ Modules: A Case Study - CppCon 2022**,” October 20, 2022. Accessed April 18, 2023. <https://www.youtube.com/watch?v=0f5N1-JKo4D4&t=1s>.
- Gabriel Dos Reis and Cameron DaCamara, “**Implementing C++ Modules: Lessons Learned, Lessons Abandoned**,” December 18, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=90WGgkuyFV8>.
- Steve Downey, “**Writing a C++20 Module**,” July 5, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=A04piAqV9mg>.
- Daniela Engert, “**The Three Secret Spices of C++ Modules**,” July 1, 2021. Accessed April 18, 2023. https://www.youtube.com/watch?v=l_83lyxWGtE.
- Sy Brand, “**C++ Modules: Year 2021**,” May 6, 2021. Accessed April 18, 2023. <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 April 18, 2023. <https://www.youtube.com/watch?v=j4du4LNsLiI>.
- Marc Gregoire, “**C++20: An (Almost) Complete Overview**,” September 26, 2020. Accessed April 18, 2023. <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 April 18, 2023. <https://www.youtube.com/watch?v=ow2zvOUdd9M>.
- Timur Doumler, “**How C++20 Changes the Way We Write Code**,” October 10, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=ImLF1LjSveM>.
- Daniela Engert, “**Modules: The Beginner’s Guide**,” May 2, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Kqo-jIq4V3I>.
- Bryce Adelstein Lelbach, “**Modules Are Coming**,” May 1, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=yee9i2rUF3s>.
- Pure Virtual C++ 2020 Conference, April 30, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=c1ThUFISDF4>
- “**Demo: C++20 Modules**,” March 30, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=6SKIUeRaLZE>.
- Daniela Engert, “**Dr Module and Sister #include**,” December 5, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=0CFOT1e2G-A>.
- Boris Kolpackov, “**Practical C++ Modules**,” October 18, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=szHV6RdQdg8>.
- Michael Spencer, “**Building Modules**,” October 6, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=L0SHHkBenss>.
- Gabriel Dos Reis, “**Programming with C++ Modules: Guide for the Working Software Developer**,” October 5, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=tjSu-KOz5HK4>.
- Nathan Sidwell, “**Converting to C++20 Modules**,” October 4, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=KVswIEw3TTw>.

- Gabriel Dos Reis, “C++ Modules: What You Should Know,” September 13, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=MP6SJEBt6Ss>
- Richárd Szalay, “The Rough Road Towards Upgrading to C++ Modules,” June 16, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=XJxQs8qgn-c>.
- Nathan Sidwell, “C++ Modules and Tooling,” October 4, 2018. Accessed April 18, 2023. https://www.youtube.com/watch?v=4y0Z8Zp_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 April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <https://wg21.link/P2409R0>.
- Andreas Fertig, *Programming with C++20: Concepts, Coroutines, Ranges, and More*, 2021. <https://andreasfertig.info/books/programming-with-cpp20/>.
- Nathan Sidwell, “C++ Modules: A Brief Tour,” October 19, 2020. Accessed April 18, 2023. <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 April 18, 2023. <https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/>.
- Vassil Vassilev, David Lange, Malik Shahzad Muzaffar, Mircho Rodozov, Oksana Shadura and Alexander Penev, “C++ Modules in ROOT and Beyond,” August 25, 2020. Accessed April 18, 2023. <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 April 18, 2023. <https://www.stroustrup.com/hop120-main-p5-p-bfc9cd4--final.pdf>.
- Rainer Grimm, “C++20: Further Open Questions to Modules,” June 8, 2020. Accessed April 18, 2023. <https://www.modernescpp.com/index.php/c-20-open-questions-to-modules>.
- Rainer Grimm, “C++20: Structure Modules,” June 1, 2020. Accessed April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <https://wg21.link/p2172r0>.
- Rainer Grimm, “C++20: The Advantages of Modules,” May 10, 2020. Accessed April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <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 April 18, 2023. <https://quuxplusone.github.io/blog/2019/11/07/modular-hello-world/>.
- “Understanding C++ Modules: Part 3: Linkage and Fragments,” October 7, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/10/07/modules-3.html>.
- Rainer Grimm, “More Details to Modules,” May 13, 2019. Accessed April 18, 2023. <http://modernescpp.com/index.php/c-20-more-details-to-modules>.
- Rainer Grimm, “Modules,” May 6, 2019. Accessed April 18, 2023. <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 April 18, 2023. <https://wg21.link/P1687R1>.
- Corentin Jabot, “Naming Guidelines for Modules,” June 16, 2019. Accessed April 18, 2023. <https://wg21.link/P1634R0>.
- “Understanding C++ Modules: Part 2: `export`, `import`, Visible, and Reachable,” March 31, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/03/31/modules-2.html>.
- “Understanding C++ Modules: Part 1: Hello Modules, and Module Units,” March 10, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/03/10/modules-1.html>.
- Rene Rivera, “Are Modules Fast? (Revision 1),” March 6, 2019, Accessed April 18, 2023. <https://wg21.link/p1441r1>.
- Richard Smith, “Merging Modules,” February 22, 2019. Accessed April 18, 2023. <https://wg21.link/p1103r3>.
- “C++ Modules Might Be Dead-on-Arrival,” January 27, 2019. Accessed April 18, 2023. <https://vector-of-bool.github.io/2019/01/27/modules-doa.html>.
- Richard Smith and Gabriel Dos Reis, “Merging Modules,” June 22, 2018. Accessed April 18, 2023. <https://wg21.link/p1103r0>.
- Gabriel Dos Reis and Richard Smith, “Modules for Standard C++,” May 7, 2018. Accessed April 18, 2023. <https://wg21.link/p1087r0>.
- Richard Smith, “Another Take on Modules (Revision 1),” March 6, 2018. Accessed April 18, 2023. <https://wg21.link/p0947r1>.
- Bjarne Stroustrup, “Modules and Macros,” February 11, 2018. Accessed April 18, 2023. <https://wg21.link/p0955r0>.
- Dmitry Guzev, “A Few Words on C++ Modules,” January 8, 2018. Accessed April 18, 2023. <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 April 18, 2023. <https://wg21.link/n4720>.
- Gabriel Dos Reis and Pavel Curtis, “Modules, Componentization, and Transition,” October 5, 2015. Accessed April 18, 2023. <https://wg21.link/p0141r0>.

- Gabriel Dos Reis, Mark Hall and Gor Nishanov, “A Module System for C++,” May 27, 2014. Accessed April 18, 2023. <https://wg21.link/n4047>.
- Daveed Vandevoorde, “Modules in C++ (Revision 6),” January 11, 2012. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf>.

Documentation

- “[Section] 3.23 C++ Modules.” Accessed April 18, 2023. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.
- “C++20 Standard: [Section] 10 Modules.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/module>.
- “Modules—Module Partitions.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/modules>.

Appendix: Modules Glossary

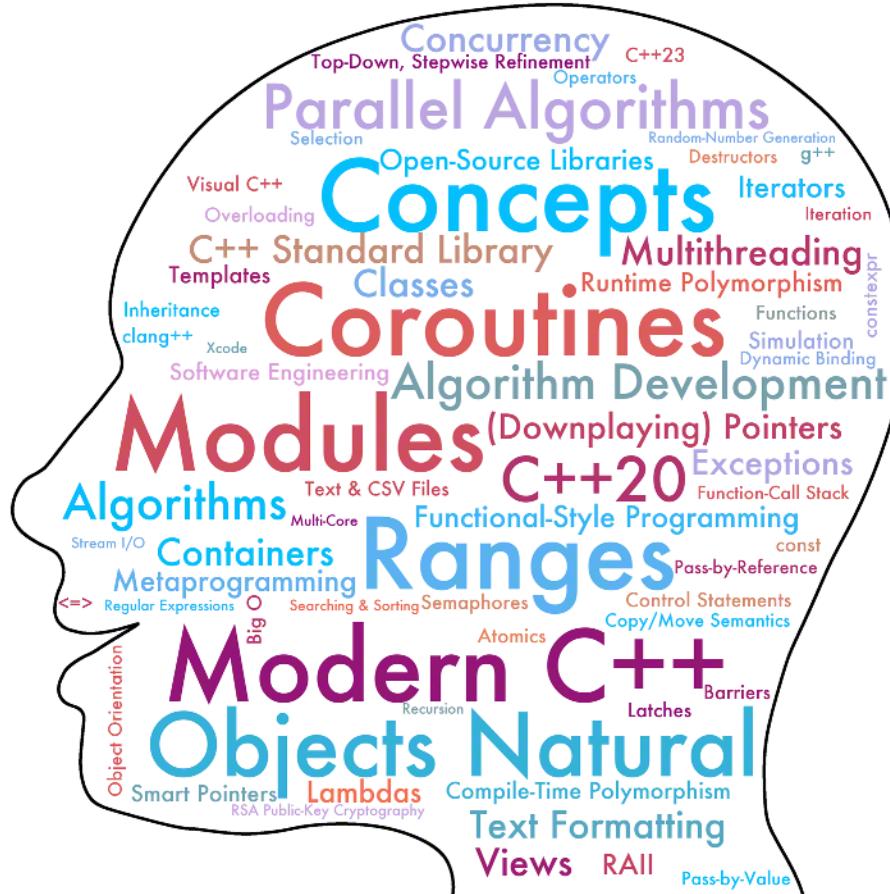
- **export a declaration**—Make a declaration available to translation units that **import** the corresponding module.
- **export a definition**—Make a definition (such as a template) available to **translation units** that **import** the corresponding module.
- **export followed by braces**—A module 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 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 in a module that are 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.
- **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.

This page intentionally left blank

Parallel Algorithms and Concurrency: A High-Level View

17



Objectives

In this chapter, you'll:

- Understand concurrency, parallelism and multithreading.
 - Use high-level concurrency features such as parallel algorithms and C++20 latches and barriers.
 - Understand the thread life cycle.
 - Use the `<chrono>` header's timing features to profile sequential and parallel algorithm performance on multi-core systems.
 - Implement correct producer-consumer relationships.
 - Synchronize access to shared mutable data by multiple threads using `std::mutex`, `std::lock_guard`, `std::condition_variable` and `std::unique_lock`.
 - Use `std::async` and `std::future` to execute long calculations asynchronously and get their results.
 - Use other C++20 concurrency features, such as semaphores and enhanced atomics.
 - Learn about possible future concurrency features.

Outline

17.1	Introduction	17.7	Producer–Consumer: Minimizing Waits with a Circular Buffer
17.2	Standard Library Parallel Algorithms	17.8	Readers and Writers
17.2.1	Example: Profiling Sequential and Parallel Sorting Algorithms	17.9	Cooperatively Canceling <code>jthreads</code>
17.2.2	When to Use Parallel Algorithms	17.10	Launching Tasks with <code>std::async</code>
17.2.3	Execution Policies	17.11	Thread-Safe, One-Time Initialization
17.2.4	Example: Profiling Parallel and Vectorized Operations	17.12	A Brief Introduction to Atomics
17.2.5	Additional Parallel Algorithm Notes	17.13	Coordinating Threads with C++20 Latches and Barriers
17.3	Multithreaded Programming	17.13.1	C++20 <code>std::latch</code>
17.3.1	Thread States and the Thread Life Cycle	17.13.2	C++20 <code>std::barrier</code>
17.3.2	Deadlock and Indefinite Postponement	17.14	C++20 Semaphores
17.4	Launching Tasks with <code>std::jthread</code>	17.15	C++23: A Look to the Future of C++ Concurrency
17.4.1	Defining a Task to Perform in a Thread	17.15.1	Parallel Ranges Algorithms
17.4.2	Executing a Task in a <code>jthread</code>	17.15.2	Concurrent Containers
17.4.3	How <code>jthread</code> Fixes <code>thread</code>	17.15.3	Other Concurrency-Related Proposals
17.5	Producer–Consumer Relationship: A First Attempt	17.16	Wrap-Up
17.6	Producer–Consumer: Synchronizing Access to Shared Mutable Data		
17.6.1	Class <code>SynchronizedBuffer</code> : Mutexes, Locks and Condition Variables		
17.6.2	Testing <code>SynchronizedBuffer</code>		

17.1 Introduction

It would be nice if we could focus our attention on performing only one task at a time and doing it well. That can be difficult to do in a complex world where so much is happening at once. This chapter presents C++’s features for building applications that create and manage multiple tasks. This can significantly improve program performance and responsiveness.

Sequential, Concurrent and Parallel Operation of Multiple Tasks

When we say that two tasks operate **sequentially**, we mean they operate one after the other “in sequence.” When we say that two tasks are operating **concurrently**, we mean that they’re both **making progress**, possibly in small increments and not necessarily simultaneously.

Let’s clarify that. Until the early 2000s, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a portion of each task before moving on to the next so that all tasks keep progressing. For example, it’s common for personal computers to concurrently perform cloud backups, compile a program, send a file to a printer, send and receive email messages and tweets, stream video and audio, download files, and more.

When we say that two tasks operate **in parallel**, we mean they’re **truly executing simultaneously**. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration, blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It’s believed that this parallelism is possible because the human brain is thought to contain billions of “processors.” Today’s multi-core computers have multiple processors that can perform tasks in parallel.



C++ Concurrency

C++ makes concurrency available to you through the language and standard libraries. Threads of execution are what happen concurrently in a program. According to the C++ standard, “a **thread of execution** (also known as a **thread**) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.”¹

Programs can have **multiple threads of execution**, each with its own function-call stack and program counter, allowing it to execute concurrently with other threads. All the threads in a given program can share application-wide resources, such as memory and files. This capability is called **multithreading**.

A problem with **single-threaded applications** that can lead to poor responsiveness is that lengthy activities must complete before others can begin. In a **multithreaded application**, threads can be distributed across multiple cores (if available), enabling tasks to truly execute in parallel, and the application can operate more efficiently.

Multithreading also can enhance performance on single-processor systems. When one thread cannot proceed (because, for example, it’s waiting for the result of an I/O operation), another can use the processor.



A Concurrent Programming Use Case: Video Streaming

We’ll discuss various **concurrent programming** applications. For example, when streaming video over the Internet, the user may not want to wait until a lengthy video finishes downloading before starting playback. Multiple threads can be used to solve this problem. One downloads the video a “chunk” at a time (we’ll refer to this thread as a **producer**), and another plays it (we’ll refer to this thread as a **consumer**). These activities can proceed concurrently. The threads are **synchronized** to avoid choppy playback. Their actions are **coordinated** so that the player thread doesn’t begin until a sufficient portion of the video is in memory to keep the player proceeding smoothly. Producer and consumer threads **share data**. We’ll show how to synchronize threads to ensure correct execution. In the case of video streaming, synchronizing threads ensures smooth viewing, even though only a portion of the video might be in memory at once.

Thread Safety

When threads share **mutable (modifiable) data**, you must ensure that they do not corrupt it, which is known as making the code **thread-safe**. Thread safety approaches include:²

- **Immutable (constant) data**—A constant object is not modifiable, so any number of threads can access a constant object at once.
- **Mutual exclusion**—You can coordinate access to shared mutable data, allowing only one thread to access the data at a time.
- **Atomic types**—You can use low-level types that automatically ensure their operations are atomic (i.e., not interruptible), so only one thread can access and possibly modify the data at a time.

1. C++ Standard, “Multi-Threaded Executions and Data Races.” Accessed April 18, 2023. https://timsong-cpp.github.io/cppwp/n4861/intro.multithread#def:thread_of_execution.

2. “Thread Safety.” Accessed April 18, 2023. https://en.wikipedia.org/wiki/Thread_safety.

- **Thread-local storage**—In code executed by multiple threads, declaring a `static` or global variable with the `storage class thread_local` indicates that each thread should have its own copy of that variable. Threads do not share variables declared `thread_local`, so these variables do not present thread-safety issues.^{3,4}

Concurrent Programming Is Complex

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find jumping between parallel trains of thought difficult. Try the following experiment to see why multithreaded programs can be challenging to write and understand. Open three books on different topics to page 1 and try reading the books concurrently. Read a few words from the first, then a few from the second, then a few from the third, then loop back and read the next few words from the first, and so on. After this experiment, you'll appreciate some of multithreading's challenges—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer to bring the text and images into focus and pushing aside the books you're not reading. And, amid all this chaos, trying to comprehend the content of the books!

C++11 and C++14: Providing Low-Level Concurrency Features

Before C++11, C++ multithreading libraries were non-standard, platform-specific extensions. C++11 introduced **standardized multithreading**. For C++11 and C++14, the C++ Standards Committee defined mostly low-level primitives. C++11 added standard library features for implementing multithreaded applications, including low-level thread synchronization primitives called **mutexes** and **locks**.⁵ C++14 added **shared mutexes** and **shared locks**.⁶ These capabilities were then used to build higher-level C++17 and C++20 features. They're also being used to implement the higher-level features coming in C++23 and later.

C++17 and C++20: Providing Convenient Higher-Level Concurrency Features

The higher-level concurrency features introduced in C++17 and C++20 are meant to simplify concurrent programming, as are features proposed for C++23, which we'll briefly introduce in Section 17.15. Higher-level concurrency features are essential because:

- They simplify concurrent programming.
- They help you find concurrency bugs in environments where it's typically impossible to reconstruct the exact circumstances in which each bug appears.
- They help you avoid common errors.
- They make your programs easier to maintain.

C++17 added 69 parallel standard library algorithms (mentioned in Sections 14.8 and 14.9).⁷ C++20 added higher-level thread-synchronization capabilities (latches, barriers and semaphores), additional parallel algorithms and coroutines (discussed in Chapter 18,

3. If you were to explicitly share pointers or references to `thread_local` variables, they could present thread-safety issues.

4. Paul E. McKenney and J. F. Bastien, "Use Cases for Thread-Local Storage," November 20, 2014. Accessed April 18, 2023. <https://wg21.link/n4324>.

5. "C++11." Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B11>.

6. "C++14." Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B14>.

7. "C++17." Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B17>.

C++20 Coroutines).⁸ The rest of this chapter introduces C++’s features for implementing multithreaded applications.

17.2 Standard Library Parallel Algorithms

Computer processing power continues to increase, but Moore’s law⁹ has essentially expired, so hardware vendors now rely on multi-core processors for better performance. In this chapter, we emphasize high-level approaches to building concurrent applications. We begin with parallel standard library algorithm overloads. These algorithms benefit from concurrent execution by taking advantage of multi-core architectures and high-performance “vector mathematics.”^{10,11} Vector operations perform the same task on many data items simultaneously using the SIMD (single instruction, multiple data) instructions provided by many CPUs and GPUs.^{12,13}



17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Section 6.12 used the `std::sort` algorithm to sort a `std::array` in a single thread of execution. Let’s compare this algorithm with its parallel overload to determine whether there’s a performance improvement. Figure 17.1 sorts 100,000,000 randomly generated `ints` stored in vectors. We use timing features from the `<chrono>` header to demonstrate the performance improvement of parallel sorting vs. sequential sorting on a multi-core system. We compiled this program in Visual C++ and ran it on a Windows 10 64-bit computer using an 8-core Intel processor. Your results likely will differ based on your hardware, operating system and compiler, and the workload on your system when you run the example. Note that this example uses the `sort` algorithms that require random-access iterators rather than a random-access range. C++20’s `std::ranges` algorithms are not yet parallelized.



```
1 // fig17_01.cpp
2 // Profiling sequential and parallel sorting with the std::sort algorithm.
3 #include <algorithm>
4 #include <chrono> // for timing operations
```

Fig. 17.1 | Profiling sequential and parallel sorting with the `std::sort` algorithm. (Part 1 of 3.)

-
8. “C++20.” Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B20>.
 9. “Moore’s Law.” Wikipedia, Wikimedia Foundation. Accessed February 10, 2022. https://en.wikipedia.org/wiki/Moore%27s_law.
 10. Billy O’Neal (Visual C++ Team blog), “Using C++17 Parallel Algorithms for Better Performance,” September 11, 2018. Accessed April 18, 2023. <https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/>.
 11. Dietmar Kuhl, “C++17 Parallel Algorithms,” October 28, 2017. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Ve8cHE9Lnfk>.
 12. C++ Standard, “General Utilities Library—Execution Policies—Unsequenced Execution Policy.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq>.
 13. “SIMD.” Accessed April 18, 2023. <https://en.wikipedia.org/wiki/SIMD>.

```

5 #include <execution> // for execution policies
6 #include <iostream>
7 #include <iterator>
8 #include <random>
9 #include <vector>
10
11 int main() {
12     // set up random-number generation
13     std::random_device rd;
14     std::default_random_engine engine{rd()};
15     std::uniform_int_distribution ints{};
16
17     std::cout << "Creating a vector v1 to hold 100,000,000 ints\n";
18     std::vector<int> v1(100'000'000); // 100,000,000 element vector
19
20     std::cout << "Filling vector v1 with random ints\n";
21     std::ranges::generate(v1, [&]() {return ints(engine);});
22
23     // copy v1 to create identical data sets for each sort demonstration
24     std::cout << "Copying v1 to vector v2 to create identical data sets\n";
25     std::vector v2{v1};
26
27     // <chrono> library features we'll use for timing
28     using std::chrono::steady_clock;
29     using std::chrono::duration_cast;
30     using std::chrono::milliseconds;
31
32     // sequentially sort v1
33     std::cout << "\nSorting 100,000,000 ints sequentially\n";
34     auto start1{steady_clock::now()}; // get current time
35     std::sort(v1.begin(), v1.end()); // sequential sort
36     auto end1{steady_clock::now()}; // get current time
37
38     // calculate and display time in milliseconds
39     auto time1{duration_cast<milliseconds>(end1 - start1)};
40     std::cout << "Time: " << (time1.count() / 1000.0) << " seconds\n";
41
42     // parallel sort v2
43     std::cout << "\nSorting the same 100,000,000 ints in parallel\n";
44     auto start2{steady_clock::now()}; // get current time
45     std::sort(std::execution::par, v2.begin(), v2.end()); // parallel sort
46     auto end2{steady_clock::now()}; // get current time
47
48     // calculate and display time in milliseconds
49     auto time2{duration_cast<milliseconds>(end2 - start2)};
50     std::cout << "Time: " << (time2.count() / 1000.0) << " seconds\n";
51 }

```

Creating a vector v1 to hold 100,000,000 ints
 Filling vector v1 with random ints
 Copying v1 to vector v2 to create identical data sets

(continued...)

Fig. 17.1 | Profiling sequential and parallel sorting with the `std::sort` algorithm. (Part 2 of 3.)

```
Sorting 100,000,000 ints sequentially
Time: 8.296 seconds

Sorting the same 100,000,000 ints in parallel
Time: 1.227 seconds
```

Fig. 17.1 | Profiling sequential and parallel sorting with the `std::sort` algorithm. (Part 3 of 3.)

Setting Up Random-Number Generation

Lines 13–15 set up the random-number capabilities we'll use in this example. We did not specify the range of random numbers, so by default, the `uniform_int_distribution` will generate integers from 0 to `std::numeric_limits<int>::max()`—the maximum `int` value on the system.

Creating the Arrays

Line 18 creates a vector to hold 100,000,000 `ints`, and line 21 uses `std::generate` to fill the vector with random `int` values. Line 25 copies the vector, so we can compare sequential and parallel sorting on vectors with identical contents.

Timing Operations with `std::chrono`

To time the sequential and parallel sorts, we'll use features from the C++ standard library's date and time utilities found in the `<chrono>` header. Lines 28–30 indicate that we're using namespace `std::chrono`'s `steady_clock`, `duration_cast` and `milliseconds`:

- We'll use a `steady_clock` object to get the time before a sorting operation starts and after it completes. An object of this type is recommended for timing operations.¹⁴ If you need the time of day, you can use a `system_clock` object, which returns the time since January 1, 1970, at midnight UTC.¹⁵
- The `duration_cast` function template converts a duration into another measurement. For example, we'll calculate the duration between two times, then call `duration_cast` to convert the result to milliseconds.
- We'll use the `std::chrono::duration` type `milliseconds` and a `duration_cast` to determine the total execution time of each `sort` call.

Sequential Sorting

Lines 33–40 test sequential sorting and time the results. Lines 34 and 36 get the current time before and after the `sort` call (line 35). Line 39 calculates the difference between the `end1` and `start1` times, then converts the result to milliseconds, which it returns as a `std::chrono::duration` object. Line 40 calls the `duration`'s `count` member function to get the sort duration in milliseconds, then divides that by 1000.0 to display the result in seconds.

14. “`std::chrono::steady_clock`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/chrono/steady_clock.

15. “`std::chrono::system_clock`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/chrono/system_clock.

Parallel Sorting with Execution Policy `std::execution::par`

Lines 43–50 test parallel sorting and time the results. Lines 44 and 46 get the current time before and after the call to `sort`'s parallel overload (line 45). Each parallel algorithm overload requires as its first parameter an **execution policy** indicating whether to parallelize a task and, if so, how to do it. Line 45 calls `sort` with the `std::execution::par` execution policy (from header `<execution>`), which indicates that the algorithm should try to execute portions of its work simultaneously on multiple cores. Section 17.2.3 overviews the four standard execution policies.

Line 49 calculates the difference between the `end2` and `start2` times, then gets the sort duration in milliseconds. Line 50 divides that by 1000.0 and displays the result in seconds. This program's output shows that **parallel execution was 6.76 times faster than sequential execution** on our system. You can see a clear performance advantage when parallel sorting a large volume of data on multiple cores.

Caution: Parallel Is Not Always Faster

You cannot simply assume that using parallel algorithms will improve performance.

Sometimes parallel algorithms actually perform worse than the corresponding sequential algorithms.¹⁶ This is especially true when processing small numbers of elements and when using non-random-access iterators. In these cases, the overhead of parallelization may outweigh the benefits of parallel performance. Microsoft's C++ standard library implementation defaults some parallel algorithms to sequential because benchmarking showed that the parallel versions ran slower for the kinds of hardware Visual C++ targets.¹⁷ For these reasons, Microsoft's parallel versions of algorithms `copy`, `copy_n`, `fill`, `fill_n`, `move`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy` and `swap_ranges` default to sequential execution. Microsoft's Billy O'Neal recommended considering parallel algorithms for tasks that process at least 2,000 items and require more than $O(n)$ time (e.g., sorting).¹⁸

17.2.2 When to Use Parallel Algorithms

To show that parallel execution might increase sort times for small datasets, we ran the program of Fig. 17.1 with vectors of random integers from 100 to 100,000,000 elements in multiples of 10 and measured the execution times in nanoseconds. We used Visual C++ on two computers:

- our everyday four-core Windows 10, 64-bit system and
- a more powerful eight-core Windows 10, 64-bit system (on which we often run processor-intensive training of machine-learning and deep-learning artificial intelligence models).

The following table shows the results from both systems. On each, parallel execution started to (barely) outperform sequential execution at 10,000 random integers. The improvements became more significant as the number of elements increased.

16. Lucian Radu Teodorescu, "A Case Against Blind Use of C++ Parallel Algorithms," February 4–7, 2021. Accessed April 18, 2023. <https://accu.org/journals/overload/29/161/teodorescu/>.

17. O'Neal, "Using C++17 Parallel Algorithms for Better Performance."

18. O'Neal, "Using C++17 Parallel Algorithms for Better Performance."

Number of elements	4-core sequential execution (in ns)	4-core parallel execution (in ns)	8-core sequential execution (in ns)	8-core parallel execution (in ns)
100	3,200	81,900	2,800	63,400
1,000	42,500	161,900	33,300	136,400
10,000	880,400	711,400	433,900	431,600
100,000	10,205,300	6,308,200	5,888,300	1,289,500
1,000,000	98,959,700	27,816,100	75,358,800	12,486,400
10,000,000	1,065,163,900	415,386,000	814,166,200	126,300,900
100,000,000	12,361,988,600	3,444,056,800	8,473,599,400	1,230,407,100

17.2.3 Execution Policies

Figure 17.1 demonstrated the `std::execution::par` execution policy. There are four standard execution policies:¹⁹

- `std::execution::seq` indicates that an algorithm must execute in a single thread. This is an object of class `std::execution::sequenced_policy`.
- `std::execution::par` indicates that an algorithm can be parallelized. This is an object of class `std::execution::parallel_policy`.
- `std::execution::par_unseq` indicates that an algorithm can be parallelized and vectorized. This is an object of class `std::execution::parallel_unsequenced_policy`. Again, vector hardware operations simultaneously perform the same task on many data items.
- `std::execution::unseq` (C++20) indicates that an algorithm can be vectorized. This is an object of class `std::execution::unsequenced_policy`.

Compilers and libraries also can provide custom execution policies.²⁰

C++ is used on a wide range of devices and operating systems, some of which do not support parallelism. These execution policies are suggestions—compilers can ignore them or handle them differently. For example, if a compiler targets hardware that does not support vectorization, the compiler might default the `par_unseq` policy to `par`. In fact, Visual C++ “implements the parallel and parallel unsequenced policies the same way, so you should not expect better performance when using `par_unseq`.²¹ Test your code and compare its sequential, parallelized and vectorized algorithm performance to determine whether it makes sense to use the parallel algorithm overloads.



19. “`std::execution::seq`, `std::execution::par`, `std::execution::par_unseq`, `std::execution::unseq`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag.

20. C++ Standard, “20.18.1 Execution Policies.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/execpol#general>.

21. O’Neal, “Using C++17 Parallel Algorithms for Better Performance.”

17.2.4 Example: Profiling Parallel and Vectorized Operations

The program of Fig. 17.2 uses `std::transform` to demonstrate parallel execution with `std::execution::par` vs. vectorized execution with `std::execution::unseq`:

- Lines 37–38 create vectors of 100 million and one billion elements, respectively, and lines 41–44 fill each with random `int` values.
- Lines 49–58 call the abbreviated function template `timeTransform` (lines 13–28) to calculate the duration of the `std::transform` operations on each vector using each execution policy. Lines 61–67 display the timing results.
- Function `timeTransform`'s first argument is the execution policy, which we pass to `std::transform` (line 21). Lines 21–22 calculate the square root of every element in `timeTransform`'s vector argument, writing the results into another vector.
- Lines 20 and 23 time the `std::transform` call, then lines 26–27 calculate and return the duration in seconds.

We compiled and ran this example using `g++ 13.1` on a MacBook Pro with an Intel processor. On our system, using the `std::execution::unseq` execution policy required approximately half the execution time of `std::execution::par`. As in Fig. 17.1, your results likely will differ based on your hardware, operating system and compiler, and the workload on your system when you run the example.

```

1 // fig17_02.cpp
2 // Performing transforms with execution policies par and unseq.
3 #include <algorithm>
4 #include <chrono> // for timing operations
5 #include <cmath>
6 #include <execution> // for execution policies
7 #include <format>
8 #include <iostream>
9 #include <random>
10 #include <vector>
11
12 // time each std::transform call and return its duration in seconds
13 double timeTransform(auto policy, const std::vector<int>& v) {
14     // <chrono> library features we'll use for timing
15     using std::chrono::steady_clock;
16     using std::chrono::duration_cast;
17     using std::chrono::milliseconds;
18
19     std::vector<double> result(v.size());
20     auto start{steady_clock::now()}; // get current time
21     std::transform(policy, v.begin(), v.end(),
22                   result.begin(), [](auto x) {return std::sqrt(x);});
23     auto end{steady_clock::now()}; // get current time
24
25     // calculate and return time in seconds
26     auto time{duration_cast<milliseconds>(end - start)};
27     return time.count() / 1000.0;
28 }
```

Fig. 17.2 | Performing transforms with execution policies `par` and `unseq`. (Part I of 2.)

```

29
30 int main() {
31     // set up random-number generation
32     std::random_device rd;
33     std::default_random_engine engine{rd()};
34     std::uniform_int_distribution ints{0, 1000};
35
36     std::cout << "Creating vectors\n";
37     std::vector<int> v1(100'000'000);
38     std::vector<int> v2(1'000'000'000);
39
40     std::cout << "Filling vectors with random ints\n";
41     std::generate(std::execution::par, v1.begin(), v1.end(),
42                  [&] () {return ints(engine);});
43     std::generate(std::execution::par, v2.begin(), v2.end(),
44                  [&] () {return ints(engine);});
45
46     std::cout << "\nCalculating square roots:\n";
47
48     // time the transforms on 100,000,000 elements
49     std::cout << std::format("{} elements with par\n", v1.size());
50     double parTime1{timeTransform(std::execution::par, v1)};
51     std::cout << std::format("{} elements with unseq\n", v1.size());
52     double unseqTime1{timeTransform(std::execution::unseq, v1)};
53
54     // time the transforms on 1,000,000,000 elements
55     std::cout << std::format("{} elements with par\n", v2.size());
56     double parTime2{timeTransform(std::execution::par, v2)};
57     std::cout << std::format("{} elements with unseq\n", v2.size());
58     double unseqTime2{timeTransform(std::execution::unseq, v2)};
59
60     // display table of timing results
61     std::cout << "\nExecution times (in seconds):\n\n"
62     << std::format("{:>13}{:>17}{:>21}\n", "# of elements",
63                  "par (parallel)", "unseq (vectorized)")
64     << std::format("{:>13}{:>17.3f}{:>21.3f}\n",
65                  v1.size(), parTime1, unseqTime1)
66     << std::format("{:>13}{:>17.3f}{:>21.3f}\n",
67                  v2.size(), parTime2, unseqTime2);
68 }

```

Creating vectors
 Filling vectors with random ints
 Calculating square roots:
 100000000 elements with par
 100000000 elements with unseq
 1000000000 elements with par
 1000000000 elements with unseq
 Execution times (in seconds):

# of elements	par (parallel)	unseq (vectorized)
100000000	2.401	1.215
1000000000	22.969	11.787

Fig. 17.2 | Performing transforms with execution policies `par` and `unseq`. (Part 2 of 2.)

17.2.5 Additional Parallel Algorithm Notes

In addition to adding parallel overloads for 69 existing algorithms, C++17 added seven new parallel algorithms:

- **for_each_n**²² applies a function to the first n elements of a range.
- **exclusive_scan**²³ and **inclusive_scan**²⁴ are parallelized versions of algorithm **partial_sum** (introduced in Section 14.4.13). The parallel algorithms differ in that **exclusive_scan** does not include the n th input element when calculating the n th sum, but **inclusive_scan** does. Like **partial_sum**, these parallel algorithms can be customized to use binary operations other than addition.
- **transform_exclusive_scan**²⁵ and **transform_inclusive_scan**²⁶ are the same as **exclusive_scan** and **inclusive_scan**, respectively, but apply a transformation function to each element before calculating the sums (or other binary operations).
- **reduce** produces a single value from a range of values (e.g., calculating the sum of a container’s elements or finding a container’s maximum value).
- **transform_reduce** performs a transformation on each element in a range or each pair of elements in a pair of ranges, then reduces the results to a single value.

Parallel Algorithm Names

Some parallel algorithms have different names from their sequential equivalents. For example, the **reduce** algorithm is the parallel equivalent of the **accumulate** algorithm.²⁷

 Unlike **accumulate**, **reduce** does not guarantee the order in which elements are processed, which enables **reduce** to be parallelized for better performance.²⁸

Restrictions

Unless the algorithm’s documentation specifies otherwise, the functions, lambdas or function objects you pass to algorithms must not modify any objects referred to directly or indirectly by their arguments.²⁹ Also, the standard library algorithms might copy their function-object arguments, so function objects passed to standard library algorithms should not maintain internal state information that could be incorrect if the function objects are copied.³⁰

-
22. “`std::for_each_n`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/for_each_n.
 23. “`std::exclusive_scan`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/exclusive_scan.
 24. “`std::inclusive_scan`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/inclusive_scan.
 25. “`std::transform_exclusive_scan`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan.
 26. “`std::transform_inclusive_scan`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan.
 27. Dietmar Kuhl, “C++17 Parallel Algorithms,” October 28, 2017. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Ve8cHE9Lnfk>.
 28. Sy Brand, “`std::accumulate` vs. `std::reduce`,” May 15, 2018. Accessed April 18, 2023. <https://blog.tartanllama.xyz/accumulate-vs-reduce/>.
 29. C++ Standard, “25.3.2 Requirements on User-Provided Function Objects.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/algorithms.parallel#user>.
 30. C++ Standard, “25.2 Algorithms Requirements.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/algorithms.requirements#10>.





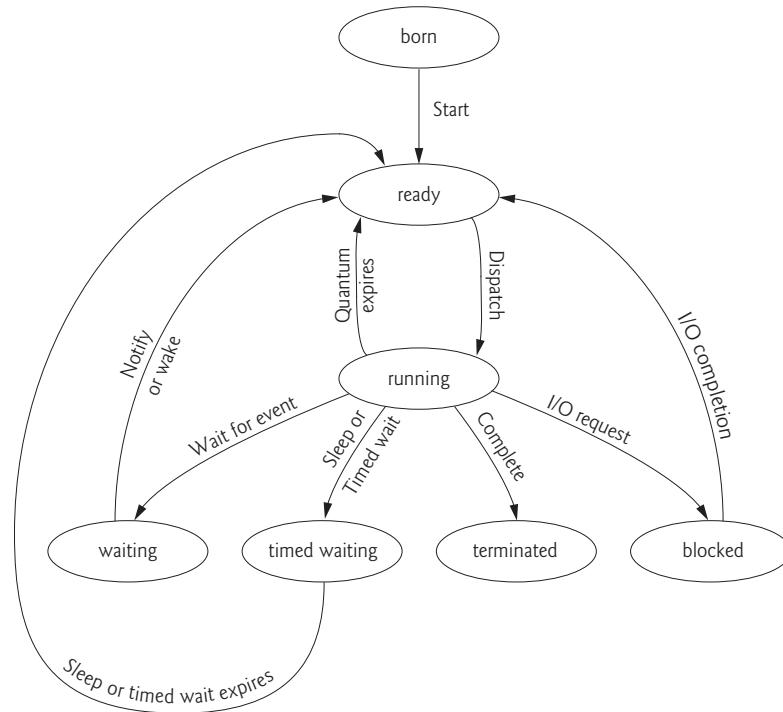
17.3 Multithreaded Programming

In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, enabling your program to complete faster. To take full advantage of multi-core architecture, you need to write multithreaded applications. When a program's tasks are split into separate threads, a multi-core system can run those threads in parallel when sufficient cores are available.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other applications and other activities the operating system runs on your behalf. All kinds of tasks are typically running in the background on your system. When you run our examples, the time to perform each task will vary based on your computer's processor speed, the number of cores and what's running on your computer. It's not unlike driving to the supermarket. The time it takes can vary based on traffic conditions, weather, presence of emergency vehicles and other factors. Some days the drive might take 10 minutes, but it could take longer, for example, during rush hour or bad weather.

17.3.1 Thread States and the Thread Life Cycle

At any time, a thread is said to be in one of several **thread states**. Here's a general-purpose thread-state switching diagram we borrowed from our operating systems book:³¹



³¹. Harvey Deitel, Paul Deitel and David Choffnes, "Chapter 4, Thread Concepts." *Operating Systems*, 3/e, p. 153. Upper Saddle River, NJ: Prentice Hall, 2004.

This diagram should give you a sense of the kinds of state-switching operations going on “under the hood.” Several of these terms are discussed later. C++’s concurrency primitives hide much of this complexity, greatly simplifying multithreaded programming and making it less error-prone.

Born and Ready States

A new thread begins its life cycle in the **born state** and remains in this state until the program starts the thread, which moves it to the **ready state**. In C++, constructing a thread object with a function as an argument (shown in Section 17.4) creates a thread and immediately starts it, making it **ready** to begin performing the task represented by that function.

Running State

A **ready** thread enters the **running state** (i.e., begins executing) when the operating system assigns it to a processor—also known as **dispatching the thread**. Operating systems give each thread a small amount of processor time—called a **quantum** or **timeslice**—to make progress on its task. Deciding how large the quantum should be is a key topic in operating systems design. When its quantum expires, the thread returns to the **ready state**, and the operating system assigns another thread to the processor. Transitions between the **ready** and **running states** are handled solely by the operating system. The process an operating system uses to determine which thread to dispatch and when is called **thread scheduling**.

Perf Scheduling decisions must be made carefully to ensure good performance and avoid problems like **indefinite postponement** of waiting threads (discussed in Section 17.3.2).

Waiting State

Sometimes a **running** thread transitions to the **waiting state** to wait for another thread to perform a task. A **waiting** thread transitions back to the **ready state** when another thread **notifies it to continue executing**.

Timed Waiting State

A **running** thread can enter the **timed waiting state** for a specified time interval. It transitions back to the **ready state** when that time interval expires or the event it’s waiting for occurs. **Timed waiting** threads and **waiting** threads cannot use a processor, even if one is available.

A **running** thread can transition to the **timed waiting state** if it provides an optional time interval when waiting for another thread to perform a task. Such a thread returns to the **ready state** when notified by another thread or its wait interval expires.

Putting a **runnable** thread to sleep also transitions the thread to the **timed waiting state**. A **sleeping thread** remains in that state for a period of time (called a **sleep interval**), after which it returns to the **ready state**. Threads sleep when they momentarily do not have work to perform. For example, a word processor may contain a thread that periodically saves the current document. If the thread did not sleep between successive backups, it would require a loop that continually tests whether to save the document. This would consume processor time without performing productive work, reducing system performance.

Perf In this case, it’s more efficient for the thread to specify a sleep interval equal to the period between successive saves and enter the **timed waiting state**. This thread returns to the **ready state** when its sleep interval expires, at which point it saves the document and reenters the **timed waiting state**.

Blocked State

A **running** thread transitions to the **blocked state** when it attempts to perform a task that cannot be completed immediately. It must temporarily wait until that task completes. For example, when a thread issues an I/O request, the operating system blocks the thread from executing until the I/O completes. At that point, the **blocked** thread transitions to the **ready state** to resume execution. A **blocked** thread cannot use a processor, even if one is available.

Terminated State

A **running** thread enters the **terminated state** when it completes its task.

Thread Scheduling

As we mentioned, **timeslicing** enables threads to share a processor. Without timeslicing, each thread runs to completion (unless it leaves the **running state** and enters the **waiting** or **timed waiting state**) before other threads get a chance to execute. With timeslicing, even if a thread has not finished executing when its quantum expires, the operating system takes the processor away and gives it to the next thread if one is available.

An operating system's **thread scheduler** determines which thread runs next. One simple thread-scheduler implementation ensures that threads each execute for a quantum in a **round-robin** fashion. This process continues until all threads run to completion.

Thread scheduling is platform-dependent. The behavior of a multithreaded program could vary across different C++ implementations, different hardware and different operating systems.



17.3.2 Deadlock and Indefinite Postponement

When a higher-priority thread enters the **ready state**, the operating system generally preempts the **running** thread (an operation known as **preemptive scheduling**). Depending on the operating system, a steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such **indefinite postponement** is referred to as **starvation**. Operating systems can employ a technique called **aging** to prevent starvation. As a thread waits in the **ready state**, the operating system gradually increases the thread's priority to ensure that it will eventually run.

Deadlock

Another problem related to indefinite postponement is called **deadlock**. A thread is **deadlocked** if it is waiting for a particular event that will not occur.^{32,33} In multithreaded systems, resource sharing is one of the primary goals. When resources are shared among a set of threads, with each thread maintaining **exclusive control** over particular resources allocated to it, deadlocks can develop in which some threads will never be able to complete execution. For example, deadlock occurs when a waiting thread, let's call this *thread1*, cannot proceed because it's waiting (either directly or indirectly) for another thread, let's call this *thread2*, to proceed while simultaneously *thread2* cannot proceed because it's waiting

32. S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," *Computer*, Vol. 13, No. 9, September 1980, pp. 58–78.

33. D. Zobel, "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, Vol. 17, No. 4, October 1983, pp. 6–16.

(either directly or indirectly) for *thread1* to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur. The result can be reduced system throughput and even system failure.

Four Necessary Conditions for Deadlock

Coffman, Elphick and Shoshani³⁴ proved that the following **four conditions are necessary for deadlock to exist**:

1. A resource may be acquired for the exclusive use of only one thread at a time (**mutual exclusion condition**).
2. A thread that has acquired an exclusive resource may hold that resource while the thread waits to obtain other resources (**wait-for condition**, also called the **hold-and-wait condition**).
3. Once a thread has obtained a resource, the system cannot remove it from the thread's control until the thread has finished using the resource (**no-preemption condition**).
4. Two or more threads are locked in a “circular chain” in which each thread is waiting for one or more resources that the next thread in the chain is holding (**circular-wait condition**).

Because these are necessary conditions, a deadlock implies that each must be in effect. So, disallowing any of the necessary conditions prevents deadlock from occurring. Taken together, all four conditions are necessary and **sufficient** for deadlock to exist (i.e., if they are all in place, the system is deadlocked).

Indefinite Postponement

In **indefinite postponement**, a thread that is not deadlocked could wait for an event that might never occur or might occur unpredictably far in the future due to biases in the system's resource-scheduling policies. In some cases, the price for making a system deadlock-free and indefinite-postponement-free is high. In mission-critical systems, the price must be paid no matter how high because allowing deadlock or indefinite postponement to develop could be catastrophic, especially if it puts human life at risk.

Preventing Deadlock

There are various deadlock-prevention techniques. Havender observed that a deadlock cannot occur if a system prevents any of the **four necessary conditions** and suggested several deadlock-prevention strategies.³⁵

The most practical approach is for each thread to request all its required resources at once and not proceed until all have been granted. If a thread requests and gets all its needed resources at once, runs to completion, then releases them, there cannot be a circular wait, and the thread cannot deadlock. If the thread cannot get all its resources at once, it should cancel the request and try again later, allowing other threads to proceed.

34. E. G. Coffman, Jr., M. J. Elphick and A. Shoshani, “System Deadlocks,” *Computing Surveys*, Vol. 3, No. 2, June 1971, p. 69.

35. J. W. Havender, “Avoiding Deadlock in Multitasking Systems,” *IBM Systems Journal*, Vol. 7, No. 2, 1968, pp. 74–84.

Requesting all required resources at once is not a perfect solution—it can lead to indefinite postponement and might result in poor system-resource utilization. For example, if a thread performs a 10-minute task requiring five resources,



- one for the task's entire duration and
- the others for only the task's last minute of execution,

you'll get poor utilization on the four you do not need until the last minute.

The Dangers of Waiting

Deadlock and indefinite postponement each involve some form of waiting. In concurrent programming, these waiting scenarios often develop in subtle ways that are not easily detectable, especially as the number of active concurrent tasks grows. The consequences when people's lives are at stake could be devastating. As you work with concurrency, you should develop a healthy sense of caution. Can you build correctly functioning concurrent systems? Yes, you can. Is it easy? Not always. A crucial key to building reliable business-critical and mission-critical systems is the trend toward developing and employing higher-level concurrency primitives. A significant part of this chapter is devoted to this higher-level approach, but first, we'll look at a few low-level building blocks.

17.4 Launching Tasks with `std::jthread`

The C++ standard provides two classes for launching concurrent tasks in an application—`std::thread` and `std::jthread` (C++20).³⁶ Both classes are defined in the `<thread>` header, which also includes utility functions, such as `get_id`, `sleep_for` and `sleep_until`. We use `std::jthread`^{37,38} exclusively in this chapter because it fixes several problems with `std::thread`.

To specify a task that can execute concurrently with other tasks:

- create a function, lambda or function object that defines the task to perform, then
- initialize a `std::jthread` object with the function, lambda or function object to execute it in a separate thread.

The task's return value is ignored. As you'll see, other mechanisms are used to communicate data between threads.

Exceptions in `jthreads`

If a `std::jthread`'s task exits via an exception, the program terminates by calling `std::terminate`.



17.4.1 Defining a Task to Perform in a Thread

This example consists of the header `printtask.h` (Fig. 17.3) and the main application (Fig. 17.4). We use the function `id` (Fig. 17.3, lines 11–15) in output statements to show

36. We'll also present higher-level capabilities in which you do not explicitly create threads.

37. In clang++, versions 13 and higher support `std::jthread`. To run examples that use it on earlier clang++ versions, replace `std::jthread` with `std::thread` and `join` the threads as shown in Section 17.4.3.

38. On Linux, add the `-pthread` compiler flag to your compilation commands to use `std::jthread`.

which thread is executing at a given time. Every thread has a **unique ID number**, including `main`'s thread, threads created with `std::jthread` or `std::thread` and threads created for you by library functions. In line 13, the expression

```
std::this_thread::get_id()
```

invokes the `std::this_thread` namespace's `get_id` function to get the executing thread's unique ID number, which is returned as a `std::thread::id` object. We use that object's overloaded operator`<<` to convert the ID to a `std::string`.

```
1 // Fig. 17.3: printtask.h
2 // Function printTask defines a task to perform in a separate thread.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <sstream>
7 #include <string>
8 #include <thread>
9
10 // get current thread's ID as a string
11 std::string id() {
12     std::ostringstream out;
13     out << std::this_thread::get_id();
14     return out.str();
15 }
16
17 // task to perform in a separate thread
18 void printTask(const std::string& name,
19                 std::chrono::milliseconds sleepTime) {
20
21     // <chrono> library features we'll use for timing
22     using std::chrono::steady_clock;
23     using std::chrono::duration_cast;
24     using std::chrono::milliseconds;
25
26     std::cout << std::format("{} (ID {}) going to sleep for {} ms\n",
27                             name, id(), sleepTime.count());
28
29     auto startTime{steady_clock::now()}; // get current time
30
31     // put thread to sleep for sleepTime milliseconds
32     std::this_thread::sleep_for(sleepTime);
33
34     auto endTime{steady_clock::now()}; // get current time
35     auto time{duration_cast<milliseconds>(endTime - startTime)};
36     auto difference{duration_cast<milliseconds>(time - sleepTime)};
37     std::cout << std::format("{} (ID {}) awakens after {} ms ({} + {})\n",
38                             name, id(), time.count(), sleepTime.count(), difference.count());
39 }
```

Fig. 17.3 | Function `printTask` defines a task to perform in a separate thread.

Function `printTask` (lines 18–39) implements the task we'd like to perform. The parameters represent

- a name we use to identify each task in the program’s output and
- a `sleepTime` in milliseconds that we use to force the executing thread to give up the processor for at least that amount of time.

When a thread calls `printTask`:

- Lines 26–27 display a message indicating the currently executing task’s name, the **unique thread ID** and the `sleepTime` in milliseconds.³⁹
- Line 29 gets the time before the thread goes to sleep.
- Line 32 invokes the `std::this_thread` namespace’s `sleep_for` function to put the thread to sleep for at least the specified amount of time. At this point, the thread loses the processor, possibly enabling another thread to execute. Our examples often make threads sleep to simulate performing work. We also use randomized sleeping to emphasize that you cannot predict when each thread will receive processor time to execute its task. The `std::this_thread` namespace also provides `sleep_until`, which sleeps until a specified time.
- When the thread’s sleep time expires, the thread reenters the ready state but does not necessarily start executing immediately. Eventually, when the operating system assigns a processor to the thread, line 34 gets the time, line 35 calculates the total time the thread was not executing, and line 36 calculates the difference between the total time and the `sleepTime`. Then, lines 37–38 display the times.
- When `printTask` terminates, its `jthread` enters the **terminated state**.

17.4.2 Executing a Task in a `jthread`

Figure 17.4 launches two concurrent threads that execute `printTask` (Fig. 17.3) and shows two sample outputs. Lines 14–16 set up random-number generation to choose random sleep up to 5,000 milliseconds. Line 18 creates a `vector` to store `std::jthreads`. We use this to enable `main` to wait for the threads to complete their tasks before the program terminates. We’ll say more about this momentarily.

```

1 // Fig. 17.4: printtask.cpp
2 // Concurrently executing tasks with std::jthreads.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <random>
7 #include <string>
8 #include <thread>
9 #include <vector>
10 #include "printtask.h"

```

Fig. 17.4 | Concurrently executing tasks with `std::jthreads`. (Part 1 of 2.)

39. When multiple threads display output with `std::cout`, their outputs might be interleaved, making the outputs appear corrupted. Preventing this requires thread synchronization (Section 17.6).

```

11
12 int main() {
13     // set up random-number generation
14     std::random_device rd;
15     std::default_random_engine engine{rd()};
16     std::uniform_int_distribution ints{0, 5000};
17
18     std::vector<std::jthread> threads; // stores the jthreads
19
20     std::cout << "STARTING JTHREADS\n";
21
22     // start two jthreads
23     for (int i{1}; i < 3; ++i) {
24         std::chrono::milliseconds sleepTime{ints(engine)};
25         std::string name{std::format("Task {}", i)};
26
27         // create a jthread that calls printTask, passing name and sleepTime
28         // as arguments and store the jthread, so it is not destructed until
29         // the vector goes out of scope at the end of main; each jthread's
30         // destructor automatically joins the jthread
31         threads.push_back(std::jthread{printTask, name, sleepTime});
32     }
33
34     std::cout << "\nJTHREADS STARTED\n";
35     std::cout << "\nMAIN ENDS\n";
36 }

```

```

STARTING JTHREADS
JTHREADS STARTED
MAIN ENDS
Task 2 (ID 15704) going to sleep for 4547 ms
Task 1 (ID 15624) going to sleep for 3648 ms
Task 1 (ID 15624) awakens after 3651 ms (3648 + 3)
Task 2 (ID 15704) awakens after 4555 ms (4547 + 8)

```

```

STARTING JTHREADS
JTHREADS STARTED
MAIN ENDS
Task 1 (ID 9368) going to sleep for 441 ms
Task 2 (ID 16876) going to sleep for 2614 ms
Task 1 (ID 9368) awakens after 449 ms (441 + 8)
Task 2 (ID 16876) awakens after 2618 ms (2614 + 4)

```

Fig. 17.4 | Concurrently executing tasks with `std::jthread`. (Part 2 of 2.)

Lines 23–32 create two `std::jthread` objects:

- Line 24 picks a **random sleep time**.
- Line 25 creates a **string** that we'll use to identify the task in the outputs.
- Line 31 creates each **jthread** and appends it to the **vector**. The **jthread** constructor receives the function to execute (`printTask`) and the arguments that

should be passed to that function (`name` and `sleepTime`). This statement creates the `jthread` as a **temporary object**, so the vector's `push_back` function that receives an *rvalue* reference moves the object into the new vector element.

Constructing a `jthread` with a function to execute starts the `jthread`, so the operating system can schedule it for execution on one of the system's cores. Line 34 outputs a message indicating the threads were started, and line 35 indicates that `main` ends.

Waiting for Previously Scheduled Tasks to Terminate

After scheduling tasks to execute, you typically want to wait for them to complete—for example, to use their results. You tell `main` to wait for a `jthread` to complete its task by “joining the thread” with a call to its **join function**. This can be done explicitly or, as we do in this program, implicitly via `jthread`'s **destructor**—one of `jthread`'s benefits over `std::thread` (see Section 17.4.3). As you know, when `main` ends, its local variables go out of scope, and their destructors are called. When the vector's destructor executes, each `jthread` element's destructor executes, calling that `jthread`'s **join** function. When all the joined `jthreads` have completed their execution, the program can terminate.

Main Thread

The code in `main` executes in the **main thread**. Function `printTask` executes when the operating system dispatches the corresponding `jthread` sometime after it enters the **ready state**.

Sample Outputs

The sample outputs show each task's name and sleep time as the thread goes to sleep. The **thread with the shortest sleep time typically awakens first, but that is not guaranteed**. When each thread continues, it displays its task name, **unique thread ID** and timings, then terminates. The first output shows the tasks going to sleep in a different order from that in which we created their `jthreads`. We **cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started**. This is one of the challenges of multithreaded programming.

17.4.3 How `jthread` Fixes `thread`

The C++ Core Guidelines say to prefer `std::jthread` over `std::thread`.⁴⁰ Class `thread` has various problems. As discussed in the preceding example, a `jthread` automatically joins each `thread` to ensure that its thread completes execution before the program terminates. In fact, the name `jthread` is short for “joining thread.” On the other hand, if you do not join a `thread` (or detach it) before it's destroyed, its destructor calls `std::terminate`, immediately terminating the application.⁴¹ To prevent that, you must explicitly join each `thread`. If the previous program used `threads`, we would have added a loop like the following before `main`'s closing brace:

- 40. C++ Core Guidelines, “CP.25: Prefer `gs1::joining_thread` over `std::thread`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-joining-thread>. [Note: This guideline says to prefer `std::jthread` in C++20.]
- 41. Detaching a `thread` separates it from the `thread` or `jthread`, but the operating system continues executing the thread. The C++ Core Guidelines recommend against detaching because it “makes it harder to monitor and communicate with the detached thread.” For more details, see “CP.26: Don't `detach()` a Thread.” Accessed February 10, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-detached_thread.

```
for (auto& t : threads) {
    t.join();
}
```

Err Even with the preceding loop, there's another potential error. If the function that launched the `threads` exits via an uncaught exception before joining each thread, the `threads` will be destroyed, and the first `thread` destructor to execute will call `std::terminate`.

Typically, you should always `join` (or `detach`) each `thread`, so `jthread` fixes the preceding problem by calling its own `join` function in its destructor.⁴² Of course, the destructor executes anytime a `jthread` goes out of scope:

- at the end of the block that created the `jthread` or
- when the block terminates due to an exception.

Class `jthread` also fixes other problems with `thread`. In particular, `jthread` supports **cooperative cancellation** (Section 17.9), supports proper move semantics, and is an **RAII** type (discussed in Section 11.5) that correctly cleans up the resources it uses.⁴³

17.5 Producer–Consumer Relationship: A First Attempt

As you'll soon see, when concurrent threads share mutable data and that data is modified by one or more of them, indeterminate results may occur:

- If one thread is updating a shared object and another thread also tries to update it, it's uncertain which thread's update will take effect.
- Similarly, if one thread is updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will see the old value or the new one.

In such cases, the program's behavior cannot be trusted. Sometimes the program will produce the correct results, and sometimes it will not. There won't be any indication that the shared mutable object was manipulated incorrectly. Worse yet, multithreaded code could appear to run correctly on one run and incorrectly on the next.

In this section and the next, we'll present two examples. The first demonstrates the problems with concurrent threads accessing shared mutable data. The second fixes those problems. Both demonstrate the **producer–consumer relationship** in which

- a **producer** thread generates data and stores it in a shared object and
- a **consumer** thread reads data from that shared object.

Another Example of a Producer–Consumer Relationship

Print spooling is one example of a producer–consumer relationship. A printer is an **exclusive resource**. Although a printer might not be available when you want to print from an application (the producer), you can still “complete” the print task. The data is temporarily **spooled** (that is, stored) until the printer becomes available. Similarly, when the printer (a

42. Nicolai Josuttis, “Why and How We Fixed `std::thread` by `std::jthread`,” July 29, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=e1F12Vh1H8>.

43. Nicolai Josuttis, *C++20: The Complete Guide*, Chapter 15, `std::jthread` and Stop Tokens,” November 21, 2021. <http://cppstd20.com/>.

consumer) is available, it doesn't wait until a current user wants to print. The spooled print jobs can be printed when the printer becomes available.

Synchronization

In a multithreaded producer–consumer relationship, a **producer thread** generates data, placing it in a shared object called a **buffer**. A **consumer thread** reads data from the buffer. This relationship requires **synchronization** to ensure that values are produced and consumed correctly. All operations on **shared mutable data** accessed by concurrent threads must be **guarded with a lock** to prevent corruption, as we'll show in Section 17.6.

State Dependence

Operations on the shared buffer also are **state-dependent**. The operations should proceed only if the buffer is in the correct state:

- If the buffer is in a **not-full state**, the producer may produce.
- If the buffer is in a **not-empty state**, the consumer may consume.

Similarly:

- If the buffer is in the **full state** when the producer wants to write a new value, it must **wait** until there's space in the buffer.
- If the buffer is in the **empty state** when the consumer wants to read a value, it must **wait** for new data to become available.

Logic Errors from Lack of Synchronization



Let's illustrate the dangers of concurrent threads sharing mutable data **without proper synchronization**. In Figs. 17.5–17.6, a producer thread writes the numbers 1 through 10 into a shared buffer—in this case, an `int` variable called `m_buffer` in line 23 of Fig. 17.5. A consumer thread reads this data from the shared buffer and displays the data. Though this might seem harmless, this program's output will show the errors that can occur as the unrestrained producer writes (produces) values into the shared buffer at will and the unrestrained consumer reads (consumes) those values from the shared buffer at will.

Each value the producer thread writes to the shared buffer must be consumed **exactly once** by the consumer thread. The threads in this example are not synchronized, so

- **data can be lost or garbled** if the producer places new data into the shared buffer before the consumer reads the previous data, and A small purple circle with a diagonal cross through it, followed by the word "Err".
- **data can be incorrectly duplicated** if the consumer consumes the same data again before the producer produces the next value. A small purple circle with a diagonal cross through it, followed by the word "Err".

To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. The producer thread produces values from 1 through 10. If the consumer correctly reads each value produced once and only once, the total will be 55. As you'll see, the unsynchronized producer–consumer pair can create incorrect totals (other than 55). Interestingly, the unsynchronized producer–consumer pair can incorrectly create the correct total of 55. We, of course, want to ensure correct operation. We'll show how to do that in Section 17.6.

UnsynchronizedBuffer

In Fig. 17.6, we'll share an object of class `UnsynchronizedBuffer` (Fig. 17.5) between the concurrent producer and consumer threads. `UnsynchronizedBuffer` maintains a single `int` data member (line 22). The producer thread will call the class's `put` function (lines 12–15) to place a value in the `int`, and the consumer thread will call the class's `get` function (lines 18–21) to retrieve the `int`'s value.

```

1 // Fig. 17.5: UnsynchronizedBuffer.h
2 // UnsynchronizedBuffer incorrectly maintains a shared integer that is
3 // accessed by a producer thread and a consumer thread.
4 #pragma once
5 #include <format>
6 #include <iostream>
7 #include <string>
8
9 class UnsynchronizedBuffer {
10 public:
11     // place value into buffer
12     void put(int value) {
13         std::cout << std::format("Producer writes\t{:2d}", value);
14         m_buffer = value;
15     }
16
17     // return value from buffer
18     int get() const {
19         std::cout << std::format("Consumer reads\t{:2d}", m_buffer);
20         return m_buffer;
21     }
22 private:
23     int m_buffer{-1}; // shared by producer and consumer threads
24 };

```

Fig. 17.5 | `UnsynchronizedBuffer` incorrectly maintains a shared integer that is accessed by a producer thread and a consumer thread. This class is not thread-safe.

This Buffer Is Not Protected by Synchronization

Class `UnsynchronizedBuffer` does not synchronize access to its data, so it is **not thread-safe**. Line 23 initializes the `UnsynchronizedBuffer`'s `m_buffer` member to `-1`. We use this value to show the case in which the **consumer thread attempts to consume a value before the producer thread ever places a value in `m_buffer`**. Function `put` simply assigns its argument to `m_buffer` (line 14), and function `get` simply returns `m_buffer`'s value (line 20).

Main Application

In Fig. 17.6, line 12 creates the `UnsynchronizedBuffer` object `buffer`, which stores the data shared by the concurrent producer and consumer threads. The producer thread will invoke the lambda `produce` (lines 15–36), and the consumer thread will invoke the lambda `consume` (lines 39–60). We'll discuss each lambda in more detail momentarily. Both lambdas' introducers capture `buffer` by reference, so the concurrent threads executing these lambdas share line 12's `buffer` object. Lines 62–63 display column heads for this program's output. Lines 65–66 create two `jthreads`—`producer` executes the `produce`

lambda, and `consumer` executes the `consume` lambda. Each goes out of scope at the end of `main`, which **joins** the threads. The italicized text in the output is our commentary, which is not part of the program’s output.

```

1 // Fig. 17.6: SharedBufferTest.cpp
2 // Application with concurrent jthreads sharing an unsynchronized buffer.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <random>
7 #include <thread>
8 #include "UnsynchronizedBuffer.h"
9
10 int main() {
11     // create UnsynchronizedBuffer to store ints
12     UnsynchronizedBuffer buffer;
13
14     // lambda expression that produces the values 1-10 and sums them
15     auto produce{
16         [&buffer]() {
17             // set up random-number generation
18             std::random_device rd;
19             std::default_random_engine engine{rd()};
20             std::uniform_int_distribution ints{0, 3000};
21
22             int sum{0};
23
24             for (int count{1}; count <= 10; ++count) {
25                 // get random sleep time then sleep
26                 std::chrono::milliseconds sleepTime{ints(engine)};
27                 std::this_thread::sleep_for(sleepTime);
28
29                 buffer.put(count); // set value in buffer
30                 sum += count; // add count to sum of values produced
31                 std::cout << std::format("\t{:2d}\n", sum);
32             }
33
34             std::cout << "Producer done producing\nTerminating Producer\n";
35         };
36     };
37
38     // lambda expression that consumes the values 1-10 and sums them
39     auto consume{
40         [&buffer]() {
41             // set up random-number generation
42             std::random_device rd;
43             std::default_random_engine engine{rd()};
44             std::uniform_int_distribution ints{0, 3000};
45

```

Fig. 17.6 | Application with concurrent jthreads sharing an unsynchronized buffer. (Caution: `UnsynchronizedBuffer` is *not* thread-safe.) (Part 1 of 3.)

```

46         int sum{0};
47
48         for (int count{1}; count <= 10; ++count) {
49             // get random sleep time then sleep
50             std::chrono::milliseconds sleepTime{ints(engine)};
51             std::this_thread::sleep_for(sleepTime);
52
53             sum += buffer.get(); // get buffer value and add to sum
54             std::cout << std::format("\t\t\t{:2d}\n", sum);
55         }
56
57         std::cout << std::format("\n{} {}{}\n",
58             "Consumer read values totaling", sum, "Terminating Consumer");
59     }
60 };
61
62 std::cout << "Action\tValue\tSum of Produced\tSum of Consumed\n";
63 std::cout << "-----\t-----\t-----\t-----\n";
64
65 std::jthread producer{produce}; // start producer jthread
66 std::jthread consumer{consume}; // start consumer jthread
67 }
```

Action	Value	Sum of Produced	Sum of Consumed
Producer writes	1	1	
Producer writes	2	3	— 1 lost
Consumer reads	2		2
Consumer reads	2		4 — 2 read again
Producer writes	3	6	
Consumer reads	3		7
Producer writes	4	10	
Producer writes	5	15	— 4 lost
Consumer reads	5		12
Producer writes	6	21	
Producer writes	7	28	— 6 lost
Consumer reads	7		19
Consumer reads	7		26 — 7 read again
Consumer reads	7		33 — 7 read again
Producer writes	8	36	
Producer writes	9	45	— 8 lost
Consumer reads	9		42
Producer writes	10	55	
Producer done producing			
Terminating Producer			
Consumer reads	10		52
Consumer reads	10		62 — 10 read again
 Consumer read values totaling 62			
Terminating Consumer			

Fig. 17.6 | Application with concurrent `jthreads` sharing an unsynchronized buffer. (Caution: `UnsynchronizedBuffer` is *not* thread-safe.) (Part 2 of 3.)

Action	Value	Sum of Produced	Sum of Consumed
Consumer reads	-1		-1 — reads -1 bad data (producer must go first)
Producer writes	1	1	
Producer writes	2	3	
Consumer reads	2		1 — 1 lost
Consumer reads	2		3 — 2 read again
Producer writes	3	6	
Consumer reads	3		6
Producer writes	4	10	
Consumer reads	4		10
Consumer reads	4		14 — 4 read again
Producer writes	5	15	
Consumer reads	5		19
Producer writes	6	21	
Consumer reads	6		25
Producer writes	7	28	
Consumer reads	7		32
Producer writes	8	36	
Producer writes	9	45	
Producer writes	10	55	
Producer done producing			— 8 lost
Terminating Producer			— 9 lost
Consumer reads	10		42
Consumer read values totaling 42			
Terminating Consumer			
Action	Value	Sum of Produced	Sum of Consumed
Producer writes	1	1	
Consumer reads	1		1
Producer writes	2	3	
Producer writes	3	6	
Consumer reads	3		4 — 2 lost
Producer writes	4	10	
Consumer reads	4		8
Consumer reads	4		12 — 4 read again
Consumer reads	4		16 — 4 read again
Consumer reads	4		20 — 4 read again
Producer writes	5	15	
Producer writes	6	21	
Consumer reads	6		26
Producer writes	7	28	
Producer writes	8	36	
Producer writes	9	45	
Consumer reads	9		35
Producer writes	10	55	
Producer done producing			
Terminating Producer			
Consumer reads	10		45
Consumer reads	10		55 — 10 read again
Consumer read values totaling 55			— Accidentally correct total
Terminating Consumer			

Fig. 17.6 | Application with concurrent `jthreads` sharing an unsynchronized buffer. (Caution: `UnsynchronizedBuffer` is not thread-safe.) (Part 3 of 3.)

Producer Thread

The producer thread executes the produce lambda (lines 15–36). Each loop iteration sleeps (line 27) for 0 to 3,000 milliseconds. When the thread awakens, line 29 sets the shared buffer’s value by passing count to the object’s **put** function. Lines 30–31 keep a total of the values produced so far and display that total. When the loop completes, line 34 indicates that the producer has finished producing data and is terminating. Then the lambda finishes executing, and the **producer thread terminates**. Any function called from a **jthread**’s task, such as buffer’s **put** function, executes in that thread. This fact will be important in Sections 17.6–17.7 when we synchronize the producer–consumer relationship.

Consumer Thread

The **consumer thread** executes the consume lambda (lines 39–60). Lines 48–55 iterate 10 times. Each iteration sleeps (line 51) for a random time interval of 0 to 3,000 milliseconds. Next, line 53 uses the buffer’s **get** function to retrieve the buffer’s value, then adds it to **sum**. Line 54 displays the total of the values consumed so far. When the loop completes, lines 57–58 display the sum of the consumed values, the lambda finishes executing, and the **consumer thread terminates**. Once both threads terminate, the program ends.

Random-Number Generation

Random-number generation is not thread-safe. To ensure that each thread can safely produce random numbers, we defined separate random-number generators in the lambdas **produce** (lines 18–20) and **consume** (lines 42–44) rather than sharing one random-number generator between them.⁴⁴

We Call `std::this_thread::sleep_for` Only for Demonstration Purposes

Throughout this chapter, we’ll refer to asynchronous concurrent threads. When we say “asynchronous,” we mean that the threads work pretty much independently of one another.⁴⁵ To emphasize that you cannot predict the relative speeds of asynchronous concurrent threads, we call function `std::this_thread::sleep_for` in the produce and consume lambdas. Thus, we do not know when the producer thread will write a new value or when the consumer thread will read a value. In multithreaded applications, it’s generally unpredictable when and for how long each thread will perform its task when it has a processor. These thread-scheduling issues are controlled by the operating system.

Without the `sleep_for` calls, and if the producer were to execute first, given today’s processor speeds, the producer would likely complete its task before the consumer got a chance to execute. If the consumer were to execute first, it would likely consume the same garbage data ten times, then terminate before the producer could produce the first real value.

Analyzing the Outputs⁴⁶

Recall that the **producer thread** should execute first, and every value it produces should be consumed exactly once by the **consumer thread**. We highlighted lines in the output

-
- 44. C++ Standard, “16.5.5.10 Data Race Avoidance.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/res.on.data.races>.
 - 45. Harvey Deitel, Paul Deitel and David Choffnes, Chapter 3, “Process Concepts.” *Operating Systems*, 3/e, p. 124. Upper Saddle River, NJ: Prentice Hall, 2004.
 - 46. As in Fig. 17.4, this program’s outputs can be interleaved in a manner that makes the output appear corrupted. Preventing this requires thread synchronization, which we discuss in Section 17.6.

where the producer or consumer acted out of order to emphasize the problems caused by failure to synchronize access to shared mutable data: 

- In the first output of Fig. 17.6, notice that the producer writes 1 and 2 before the consumer reads its first value (2). Therefore, the value 1 is **lost**. Later, 4, 6 and 8 are **lost**, while 2 and 10 are **read twice**, and 7 is **read three times**. So the first output produces an incorrect total of 62 instead of the correct total of 55.
- In the second output, the consumer reads the garbage value -1 **before** the producer ever writes a value. Meanwhile, 1, 8 and 9 are all **lost**, and 2 and 4 are **read twice**. The result is an incorrect consumer total of 42.
- In the third output, you see that it's possible for the **consumer** to read values that accidentally total 55 without reading every value from 1 through 10 exactly once. In this case, the values 2, 5, 7 and 8 are all **lost**, the value 4 was **read four times**, and the value 10 was **read twice**.

The outputs clearly show that access to a shared mutable object by concurrent threads must be controlled carefully; otherwise, a program will likely produce incorrect results—and perhaps even worse, could accidentally produce a “correct result” for the wrong reasons.

One challenge of multithreaded programming is spotting errors. They may occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that it's correct. This is why you should use predefined containers and higher-level primitives that handle the synchronization for you.  

To solve the problems of lost and duplicated data, the next section presents an example in which we synchronize access to the shared object, guaranteeing that each value will be processed once and only once. 

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

The errors in Section 17.5's program can be attributed to the fact that **Unsynchronized-Buffer** is not thread-safe. It allowed uncoordinated concurrent producer and consumer threads to modify and read shared mutable data, which leads to **data races** (also called **race conditions**).^{47,48} The thread that “wins the race” by getting there first performs its task, even if it should not. There's no guarantee of the order in which the concurrent producer and consumer threads will perform their tasks, resulting in cases in which:

- the producer overwrites previously written values before they're consumed, causing lost data, or

47. C++ Standard, “Data Races.” Accessed April 18, 2023. https://timsong-cpp.github.io/cppwp/n4861/intro.races#define:data_race.

48. Arthur O'Dwyer, “Back to Basics: Concurrency,” October 6, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=F6IpN7gC0sY>. If you enjoy watching videos of experts making one-hour conference presentations, check out this video. O'Dwyer heads the CppCon committee responsible for the “Back to Basics” track. He and other experts offer sessions on a broad range of topics important to C++ developers.

- the consumer reads invalid data (-1) before the producer has produced its first legitimate value or
- the consumer reads **stale values** it read previously.

The C++ standard emphasizes that “**a memory location cannot be safely accessed by two threads without some form of locking unless they are both read accesses.**”⁴⁹

Thread Synchronization, Mutual Exclusion and Critical Sections

The problems from the last example can be solved by giving **only one thread at a time** exclusive access to code that manipulates the shared mutable data. During that time, the other thread must **wait**. When the thread with exclusive access finishes accessing the shared mutable data, **the waiting thread can proceed**. This **thread synchronization** process coordinates access to shared mutable data by concurrent threads. Each thread accessing a shared object **excludes** the other thread from doing so simultaneously. This is called **mutual exclusion**. The code sections that we protect using mutual exclusion are called **critical sections**.

Executing a Set of Operations As If They Are One Operation

To make our shared buffer thread safe, we must ensure that **only one thread at a time can store a value in the buffer or read a value from the buffer**. We must also ensure that these operations cannot be divided into smaller suboperations—known as making the operations **atomic**. We do this by allowing only one thread to execute **put** or **get** at a time:

- If the producer is executing **put** when the consumer tries to execute **get**, the consumer must **wait** until the producer finishes its **put** call.
- Similarly, if the consumer is executing **get** when the producer tries to execute **put**, the producer must **wait** until the consumer finishes its **get** call.

Immutable Data Does Not Require Synchronization

The synchronization we demonstrate in the next example is required only for **shared mutable data**, which might change during its lifetime. **Immutable data** does not change, so any number of concurrent threads can access the data. The C++ Core Guidelines say, “**You can’t have a race condition on a constant**,” and for this reason, indicate that you should

- “use **const** to define objects with values that do not change after construction” and
- “use **constexpr** for values that can be computed at compile time.”⁵⁰

Doing so indicates the variables’ values will not change after they’re initialized, preventing accidental modification that could compromise thread safety. The C++ Core Guidelines

 also recommend **minimizing the use of shared mutable data to avoid data races.**⁵¹

49. “C++11 Language Extensions—Concurrency.” Accessed April 18, 2023. <https://isocpp.org/wiki/faq/cpp11-language-concurrency>.

50. C++ Core Guidelines, “Con: Constants and Immutability.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const>.

51. C++ Core Guidelines, “CP.3: Minimize Explicit Sharing of Writable Data.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data>.

17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables

Figures 17.7–17.8 demonstrate a producer thread and a consumer thread correctly accessing a `synchronized` shared mutable buffer. In this example:

- the producer always produces a value first,
- the consumer correctly consumes only after the producer produces a value and
- the producer correctly produces the next value only after the consumer consumes the previous (or first) value.

As you'll see, `SynchronizedBuffer`'s `put` and `get` functions (Fig. 17.7) handle the synchronization. We output messages from these functions for demonstration purposes only. I/O is slow compared to processor operations. So, I/O should not be performed in critical sections because it's crucial to minimize the amount of time that an object is “locked.”



```

1 // Fig. 17.7: SynchronizedBuffer.h
2 // SynchronizedBuffer maintains synchronized access to a shared mutable
3 // integer that is accessed by a producer thread and a consumer thread.
4 #pragma once
5 #include <condition_variable>
6 #include <format>
7 #include <mutex>
8 #include <iostream>
9 #include <string>
10
11 using namespace std::string_literals;
12
13 class SynchronizedBuffer {
14 public:
15     // place value into m_buffer
16     void put(int value) {
17         // critical section that requires a lock to modify shared data
18         {
19             // Lock on m_mutex to be able to write to m_buffer
20             std::unique_lock dataLock{m_mutex};
21
22             if (m_occupied) {
23                 std::cout << std::format(
24                     "Producer tries to write.\n{:<40}{}\t{}\n",
25                     "Buffer full. Producer waits.", m_buffer, m_occupied);
26
27             // wait on condition variable m_cv; the lambda in the second
28             // argument ensures that if the thread gets the processor
29             // before m_occupied is false, the thread continues waiting
30             m_cv.wait(dataLock, [&]() {return !m_occupied;});
31         }
32     }

```

Fig. 17.7 | `SynchronizedBuffer` maintains synchronized access to a shared mutable integer that is accessed by a producer thread and a consumer thread. (Part 1 of 2.)

```

33         // write to m_buffer
34         m_buffer = value;
35         m_occupied = true;
36
37         std::cout << std::format("{:<40}{}\t\t{}\n\n",
38             "Producer writes "s + std::to_string(value),
39             m_buffer, m_occupied);
40     } // dataLock's destructor releases the lock on m_mutex
41
42     // if consumer is waiting, notify it that it can now read
43     m_cv.notify_one();
44 }
45
46 // return value from m_buffer
47 int get() {
48     int value; // will store the value returned by get
49
50     // critical section that requires a lock to modify shared data
51     {
52         // lock on m_mutex to be able to read from m_buffer
53         std::unique_lock dataLock{m_mutex};

54         if (!m_occupied) {
55             std::cout << std::format(
56                 "Consumer tries to read.\n{:<40}{}\t\t{}\n\n",
57                 "Buffer empty. Consumer waits.", m_buffer, m_occupied);

58             // wait on condition variable m_cv; the lambda in the second
59             // argument ensures that if the thread gets the processor
60             // before m_occupied is true, the thread continues waiting
61             // before m_cv.wait()
62             m_cv.wait(dataLock, [&](){return m_occupied;});
63         }
64     }

65     value = m_buffer;
66     m_occupied = false;
67
68     std::cout << std::format("{:<40}{}\t\t{}\n{}\n",
69         "Consumer reads "s + std::to_string(m_buffer),
70         m_buffer, m_occupied, std::string(64, '-'));
71 } // dataLock's destructor releases the lock on m_mutex
72
73     // if producer is waiting, notify it that it can now write
74     m_cv.notify_one();
75
76     return value;
77 }
78
79 private:
80     int m_buffer{-1}; // shared by producer and consumer threads
81     bool m_occupied{false};
82     std::condition_variable m_cv;
83     std::mutex m_mutex;
84 };

```

Fig. 17.7 | SynchronizedBuffer maintains synchronized access to a shared mutable integer that is accessed by a producer thread and a consumer thread. (Part 2 of 2.)

SynchronizedBuffer's `m_buffer` and `m_occupied` Data Members

Line 80 defines the `int` data member `m_buffer` into which a producer thread will write data and from which a consumer thread will read data. The `bool` data member `m_occupied` (line 81) indicates whether `m_buffer` currently contains data. We'll use this to help keep track of the shared buffer's state for thread synchronization purposes. Variables `m_occupied` and `m_buffer` are both part of a `SynchronizedBuffer`'s state information, so you must synchronize access to both to ensure that the buffer is **thread-safe**.

SynchronizedBuffer's `std::condition_variable` Data Member

As part of synchronizing access to the buffer, we must ensure that the producer and consumer threads each do their work only when the buffer is in an appropriate state. We need a way to allow the threads to **wait, depending on certain conditions**, which we maintain via `m_occupied`:

- The producer can place a new item in the buffer only if the **buffer is not full**—that is, `m_occupied` is `false`.
- The consumer can read an item from the buffer only if the **buffer is not empty**—that is, `m_occupied` is `true`.

If a given condition is `true`, the corresponding thread may proceed. If it's `false`, the corresponding thread must wait until it becomes `true`.

We also need a way to **let a waiting thread know when conditions have changed** so it can proceed:

- If the consumer is waiting to read and the producer writes a new value into the buffer, the **buffer is now full**, so the producer should **notify the waiting consumer** that it can read that value.
- If the producer is waiting to write and the consumer reads the buffer's current value, the **buffer is now empty**, so the consumer should **notify the waiting producer** that it can write into the buffer.

These wait and notify capabilities are provided by a `std::condition_variable`⁵² (from header `<condition_variable>`). Line 82 defines the `m_cv` data member of this type.

SynchronizedBuffer's `std::mutex` Data Member

A common way to implement **mutually exclusive** access to shared mutable resources is by creating **critical sections**. These **synchronized blocks of code** execute atomically using features from the `<mutex>` header. A `std::mutex` can be owned by **only one thread at a time**. A thread that requires exclusive access to a resource must first **acquire a lock** on a `mutex`—typically at the beginning of a block of code. Other threads attempting to perform operations that require the same `mutex` will be **blocked** until the first thread **releases the lock**—typically at the end of a block of code. At that point, the **blocked** threads may attempt to acquire the lock and proceed with the operation.⁵³

52. “`std::condition_variable`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/condition_variable.

53. From a December 6, 2021, email correspondence with Anthony Williams: “There is no requirement on which thread gets the `mutex` lock when a `mutex` is unlocked. ... it is often most efficient from an OS perspective to allow the thread that has been waiting the **shortest time** to ... acquire the `mutex` lock, as its data still might be in cache.”

By placing operations in a **critical section**, we allow only one thread at a time to acquire the lock and perform the operations. If multiple **critical sections** are synchronized with the same **mutex**, only one can execute at a time. Line 83 defines **mutex** data member **m_mutex**, which we'll use in conjunction with **m_cv** to protect the critical sections of code in **SynchronizedBuffer**'s **put** and **get** functions that access the class's shared mutable data. For data that requires **mutually exclusive access**, the C++ Core Guidelines recommend defining the data together with the **mutex** used to protect it. For example, we will protect **SynchronizedBuffer**'s data members by using a **mutex** that's also a data member of **SynchronizedBuffer**.⁵⁴

CG In multithreaded programs, place all accesses to shared mutable data inside critical sections that synchronize on the same `std::mutex`. All operations within that critical section represent an atomic operation. Promptly release the lock when it's no longer needed.

SynchronizedBuffer Member Functions

Functions **put** (lines 16–44) and **get** (lines 47–78) provide synchronized access to the shared data members **m_occupied** and **m_buffer**. Only one thread at a time can enter either of these functions' critical sections on a particular **SynchronizedBuffer** object. Variable **m_occupied** is used in **put** and **get** to determine whether it's the producer's turn to write or the consumer's turn to read:

- If **m_occupied** is `false`, **m_buffer** is empty, so the producer can place a value into **m_buffer**, but the consumer cannot read **m_buffer**'s value.
- If **m_occupied** is `true`, **m_buffer** is full, so the consumer can read **m_buffer**'s value, but the producer cannot place a value into **m_buffer**.

Member Function **put** and the Producer Thread

You'll synchronize access to **SynchronizedBuffer**'s shared mutable data using the class's **condition_variable**, **mutex** and a **std::unique_lock**, which among its capabilities can lock a **mutex** that's used in conjunction with a **condition_variable**. When the producer thread invokes **put**, it must first acquire **m_mutex**'s lock to ensure that it has exclusive access to **SynchronizedBuffer**'s shared mutable data. You acquire a **mutex**'s lock by creating a **lock object** and initializing it with the **mutex** (line 20). If the **mutex**'s lock is not available, the thread creating the lock object is **blocked** and must **wait** until it can acquire the lock. Once the thread acquires the lock, the rest of that block is said to be **guarded** by the **mutex**.

As you'll see momentarily, a **unique_lock** can **release** a **mutex**'s lock when it's not a given thread's turn to perform its task, then can **reacquire** the lock later. This capability is important. Holding a lock on the **SynchronizedBuffer**'s **mutex** when a thread cannot perform its task could cause **deadlock**.

Acquiring the Lock and Checking Whether It's the Producer's Turn

When the **m_mutex**'s lock is available, line 20 **acquires** the lock. Then, line 22 checks whether **m_occupied** is `true`. If so, **m_buffer** is full, and the **producer thread must wait** until **m_buffer** is empty, so lines 23–25 output a message indicating that

- the producer thread is trying to write a value,

54. C++ Core Guidelines, "CP.50: Define a mutex Together with the Data It Guards." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-mutex>.

- `m_buffer` is full and
- the producer thread is waiting until there's space.

Waiting on the Condition Variable

Condition variables can be used to make a thread `wait` while a condition is not satisfied then to `notify` a waiting thread to proceed when a condition is satisfied:

- If the producer thread obtains the `mutex`'s lock, but the buffer is full, the thread calls `condition_variable`'s `wait` function (line 30), passing the `unique_lock` (`dataLock`) as the first argument. This causes `dataLock` to release the `mutex`'s lock, places the producer thread in `condition_variable` `m_cv`'s waiting state and removes the thread from contention for a processor. This is important because the producer thread cannot currently perform its task. So, the consumer should be allowed to access the `SynchronizedBuffer` to allow the condition (`m_occupied`) to change. Now the consumer can attempt to acquire the lock on `m_mutex`.
- When the consumer thread executing the critical section in the `get` function satisfies the condition on which the producer thread may be waiting—that is, the consumer reads the buffer's value, so the buffer is empty—the consumer thread calls `notify_one` on `m_cv` (line 75). This allows the producer thread waiting on that condition to transition to the ready state.⁵⁵ When the operating system moves the producer thread to the running state, it can attempt to reacquire the lock.⁵⁶

Once the producer is notified and implicitly reacquires `m_mutex`'s lock, `put` continues executing with the next statement after the `wait` call.

Spurious Wakeup

Occasionally, the system could move a waiting thread back into the ready state before that thread can perform its task—this is known as a **spurious wakeup**.^{57,58} The `wait` function's second argument in line 30 is a no-argument lambda that checks whether it's the producer thread's turn to access `m_buffer`. When the producer thread gets the processor, it first reacquires the `m_mutex`'s lock, then calls this lambda. If the lambda returns `false`, the thread releases the lock and returns to `condition_variable` `m_cv`'s waiting state; otherwise, the producer thread continues executing.

Updating the `SynchronizedBuffer`'s State

Line 34 assigns the produced value to `m_buffer`. Line 35 sets `m_occupied` to `true` to indicate that `m_buffer` now contains a value (i.e., a consumer can read the value, but a pro-

55. From a December 6, 2021, email correspondence with Anthony Williams: “Condition variables do not specify which thread is woken when `notify_one` is called. It is valid for *all* the waiting threads to be woken. ... In particular, it is often most efficient from an OS perspective to allow the thread that has been waiting the shortest time to be woken ..., as its data might still be in cache.”

56. In some applications, when a thread reacquires the lock, it still might not be able to perform its task—in which case, it will reenter the condition variable's waiting state and implicitly release the lock.

57. Marius Bancila, *Modern C++ Programming Cookbook*, 2/e, p. 422. Birmingham, UK: Packt Publishing, 2020.

58. “Spurious Wakeup.” Accessed April 18, 2023. https://en.wikipedia.org/wiki/Spurious_wakeup.

ducer cannot yet put another value there). Lines 37–39 indicate that the producer is writing a value into the `m_buffer`.

Releasing a Lock

At this point, execution reaches line 40—the end of the block that defines `put`'s critical section. When a thread finishes using a shared object that's managed with `unique_lock`, the thread must release the lock by calling the lock's `unlock` function implicitly or explicitly. Lock objects use RAI (discussed in Section 11.5). If you do not explicitly call `unlock`, class `unique_lock`'s destructor will implicitly call it when the lock goes out of scope at the end of the critical section. If the consumer thread was previously blocked while attempting to enter `get`'s critical section, which is guarded by the same `mutex`, the consumer thread can now acquire the lock to proceed.

Notifying the Consumer Thread to Continue

Now that the buffer is full, the producer thread calls `m_cv`'s `notify_one` function (line 43) to indicate that the buffer's state has changed. If the consumer is waiting on this condition variable, it enters the ready state and becomes eligible to reacquire the lock. Function `notify_one` returns immediately, then `put` returns to its caller (i.e., the producer thread).



For performance, you should unlock the `unique_lock` before calling `notify_one` (or `notify_all`) to ensure that the notified waiting thread does not need to wait for the `mutex` lock to become available.⁵⁹

Member Function `get` and the Consumer Thread

Functions `get` and `put` are implemented similarly. When the consumer thread invokes `get`, line 53 creates a `unique_lock` and attempts to acquire `m_mutex`'s lock. If the lock is not available (e.g., the producer has not yet released it), the consumer thread is blocked until the lock becomes available. If it is available, line 53 acquires it, so the consumer thread now owns the lock. Next, line 55 checks whether `m_occupied` is false. If so, `m_buffer` is empty, so lines 56–58 indicate that

- the consumer thread is trying to read a value,
- the buffer is empty and
- the consumer thread is waiting.

Waiting on the Condition Variable

Line 63 invokes the `m_cv`'s `wait` function to place the consumer thread in that `condition_variable`'s waiting state. Again, `wait` causes `dataLock` to release `m_mutex`'s lock, so the producer thread can attempt to acquire it and do its work.

The consumer thread remains in the waiting state until it's notified by the producer thread to proceed. At that point, the consumer thread returns to the ready state. When the operating system moves the thread to the running state, the consumer thread attempts to implicitly reacquire `m_mutex`'s lock. If it's available, the consumer thread reacquires it and `get` continues executing with the next statement after `wait`. The second argument to `m_cv`'s `wait` function is a lambda that checks whether it's the consumer thread's turn. If

59. Bancila, *Modern C++ Programming Cookbook*, p. 420.

the consumer thread gets a processor and this lambda returns `false`, the thread will return to `m_cv`'s **waiting state**.

Updating the `SynchronizedBuffer`'s State and Notifying the Producer to Continue

Line 66 stores the `value` that will be returned to the calling thread, line 67 sets `m_occupied` to `false` to indicate that `m_buffer` is now empty (i.e., the producer can produce), and lines 69–71 indicate the consumer is reading a value. At this point, `dataLock` goes out of scope, and its **destructor** releases `m_mutex`'s lock. Next, line 75 invokes `m_cv`'s **notify_one** function. If a producer thread is in `m_cv`'s **waiting state**, it enters the **ready state** and becomes eligible to **reacquire** `m_mutex`'s lock. Function **notify_one** returns immediately, then `get` returns `value` to its caller.

Pairing Waits and Notifications

When concurrent threads manipulate a shared object using locks on a given `mutex`, ensure that if one thread calls function `wait` to enter a `condition_variable`'s **waiting state**, a separate thread eventually will call `condition_variable` function `notify_one` (or `notify_all`) to transition the waiting thread back to the **ready state**. 

17.6.2 Testing `SynchronizedBuffer`

Figure 17.8 is similar to Fig. 17.6. Lines 19–24 define the lambda `getSleepTime`, which uses a `std::mutex` (line 16) and a `std::lock_guard` (line 21) to ensure synchronized access to a random-number generator that we'll share among this example's threads. We do this only for demonstration purposes in this example to show using a `lock_guard` to protect a shared resource that does not require a `condition_variable`. When you construct a `lock_guard`, it **acquires** its `mutex` argument's lock if it's available; otherwise, the thread creating the `lock_guard` is blocked until it can acquire the lock. Unlike `unique_lock`, which can release and reacquire a `mutex`'s lock—capabilities we need to use `condition_variables`—a `lock_guard` owns a `mutex`'s lock until the `lock_guard` goes out of scope. At that point, its **destructor** releases the `mutex`'s lock.⁶⁰ Thus, only one thread at a time can execute `getSleepTime`'s block. For threads that need to access resources guarded by separate `mutexes`, `std::scoped_lock` acquires locks on several `mutexes` at once and releases them all when the `scoped_lock` goes out of scope at the end of its enclosing block.⁶¹

Line 27 creates the `SynchronizedBuffer` that we share between the concurrent producer and consumer threads. Lines 30–44 and 47–61 define the `produce` and `consume` lambdas that will be called by the producer and consumer `jthreads` (lines 67–68). Each lambda captures `buffer` and `getSleepTime` by reference. Lines 63–65 display the output's column heads. When `main` ends, the `jthreads` go out of scope, automatically joining the threads, enabling them to complete execution before the program terminates.

60. “`std::lock_guard`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/lock_guard.

61. “`std::scoped_lock`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/scoped_lock.

```

1 // Fig. 17.8: SharedBufferTest.cpp
2 // Concurrent threads correctly manipulating a synchronized buffer.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <mutex>
7 #include <random>
8 #include <thread>
9 #include "SynchronizedBuffer.h"
10
11 int main() {
12     // set up random-number generation
13     std::random_device rd;
14     std::default_random_engine engine{rd()};
15     std::uniform_int_distribution ints{0, 3000};
16     std::mutex intsMutex;
17
18     // lambda for synchronized random sleep time generation
19     auto getSleepTime{
20         [&] () {
21             std::lock_guard lock{intsMutex};
22             return std::chrono::milliseconds{ints(engine)};
23         }
24     };
25
26     // create SynchronizedBuffer to store ints
27     SynchronizedBuffer buffer;
28
29     // lambda expression that produces the values 1-10 and sums them
30     auto produce{
31         [&buffer, &getSleepTime]() {
32             int sum{0};
33
34             for (int count{1}; count <= 10; ++count) {
35                 // get random sleep time then sleep
36                 std::this_thread::sleep_for(getSleepTime());
37
38                 buffer.put(count); // set value in buffer
39                 sum += count; // add count to sum of values produced
40             }
41
42             std::cout << "Producer done producing\nTerminating Producer\n";
43         }
44     };
45
46     // lambda expression that consumes the values 1-10 and sums them
47     auto consume{
48         [&buffer, &getSleepTime]() {
49             int sum{0};
50

```

Fig. 17.8 | Concurrent threads correctly manipulating a synchronized buffer. (Part 1 of 3.)

```

51         for (int count{1}; count <= 10; ++count) {
52             // get random sleep time then sleep
53             std::this_thread::sleep_for(getSleepTime());
54
55             sum += buffer.get(); // get buffer value and add to sum
56         }
57
58         std::cout << std::format("\n{} {} \n{}\n",
59             "Consumer read values totaling", sum, "Terminating Consumer");
60     }
61 };
62
63 std::cout << std::format("{:<40}{}\\t\\t{}\\n{:<40}{}\\t\\t{}\\n\\n",
64     "Operation", "Buffer", "Occupied",
65     "-----", "-----", "-----");
66
67 std::jthread producer{produce}; // start producer thread
68 std::jthread consumer{consume}; // start consumer thread
69 }
```

Operation	Buffer	Occupied
-----	-----	-----
Consumer tries to read. Buffer empty. Consumer waits.	-1	false
Producer writes 1	1	true
Consumer reads 1	1	false
-----	-----	-----
Producer writes 2	2	true
Producer tries to write. Buffer full. Producer waits.	2	true
Consumer reads 2	2	false
-----	-----	-----
Producer writes 3	3	true
Consumer reads 3	3	false
-----	-----	-----
Consumer tries to read. Buffer empty. Consumer waits.	3	false
Producer writes 4	4	true
Consumer reads 4	4	false
-----	-----	-----
Producer writes 5	5	true
Consumer reads 5	5	false
-----	-----	-----

Fig. 17.8 | Concurrent threads correctly manipulating a synchronized buffer. (Part 2 of 3.)

Producer writes 6	6	true
Producer tries to write. Buffer full. Producer waits.	6	true
Consumer reads 6	6	false

Producer writes 7	7	true
Producer tries to write. Buffer full. Producer waits.	7	true
Consumer reads 7	7	false

Producer writes 8	8	true
Producer tries to write. Buffer full. Producer waits.	8	true
Consumer reads 8	8	false

Producer writes 9	9	true
Consumer reads 9	9	false

Producer writes 10	10	true
Producer done producing Terminating Producer		
Consumer reads 10	10	false

Consumer read values totaling 55		
Terminating Consumer		

Fig. 17.8 | Concurrent threads correctly manipulating a synchronized buffer. (Part 3 of 3.)

Analyzing the Output

Study the output in Fig. 17.8 and observe that:

- Every integer produced is consumed exactly once—**no values are lost, and no values are consumed more than once**.
- The synchronization ensures the producer produces a value only when the buffer is empty, and the consumer consumes only when the buffer is full.
- The producer always produces a value before the consumer is allowed to consume a value for the first time.
- The consumer waits if the producer has not produced since the consumer last consumed.
- The producer waits if the consumer has not yet consumed the value that the producer most recently produced.

Execute this program several times to confirm that **every integer produced is consumed exactly once**. In the sample output, we applied bold to the lines indicating when the producer and consumer must wait to perform their respective tasks.

Note Regarding Output Statements in Our Synchronization Examples

In addition to performing the actual operations that manipulate the `SynchronizedBuffer`, our `synchronized put` and `get` functions print messages to the console indicating the threads' progress as they execute these functions. We do this so the messages will print in the correct order, showing that `put` and `get` are correctly synchronized. We continue to output messages from critical sections in later examples for demonstration purposes only. The C++ Core Guidelines say to minimize the duration of critical sections⁶²—that is, the amount of time an object is “locked.” Avoid performing I/O, lengthy calculations and operations that do not require synchronization while holding a lock.

Also, for demonstration purposes, lines 36 and 53 call `sleep_for` to emphasize the unpredictability of thread scheduling. Though we are not doing so here, it's important to note that a thread should never sleep while holding a lock in a real application.



17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer

The program in Section 17.6 used thread synchronization to guarantee that two concurrent threads correctly manipulated data in a shared buffer. However, the application may not perform optimally. If the threads operate at different speeds, one will spend more (or most) of its time waiting. For example:

- If the producer thread produces values faster than the consumer can consume them, the producer thread **waits** because there are no empty locations for writing.
- If the consumer consumes values faster than the producer produces them, the consumer **waits** until the producer places the next value in the shared buffer.

Even threads that operate at approximately the same relative speeds can occasionally become “out of sync” over a period of time, causing one of them to **wait** for the other.

We cannot predict the relative speeds of asynchronous concurrent threads. Interactions with the operating system, the network, the user and other components can cause threads to operate at different and ever-changing speeds. When this happens, threads wait. When threads wait excessively, programs can become less efficient, interactive programs can become less responsive, and applications can suffer potentially long delays.



Circular Buffers

Using a **circular buffer**, we can minimize **waiting** among concurrent threads that share resources and operate at the same average speeds. Such a buffer provides a fixed number of cells into which the producer can write values and from which the consumer can read those values. Internally, a circular buffer manages the producer's writes into the buffer and the consumer's reads from the buffer elements in order, beginning at the first cell and moving toward the last. When the circular buffer reaches its last element, it “wraps around” to the first element and continues from there—thus the name “circular.”

An example of the producer–consumer relationship that uses a circular buffer is the video streaming we discussed in Section 17.1. With a circular buffer:

62. C++ Core Guidelines, “CP.43: Minimize Time Spent in a Critical Section.” Accessed April 18, 2023.
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-time>.

- If the producer temporarily operates faster than the consumer, the producer can write additional values into the extra buffer cells, if any are available. This enables the producer to keep busy even though the consumer is not ready to retrieve the current value being produced.
- If the consumer temporarily operates faster than the producer, the consumer can read additional values (if there are any) from the buffer. This enables the consumer to keep busy even though the producer is not ready to produce additional values.

Even with a **circular buffer**, a producer thread could fill the buffer, forcing the producer to **wait** until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty, a consumer must **wait** until the producer produces another value.

Even a **circular buffer** is inappropriate if the producer and the consumer operate consistently at significantly different average speeds:

- If the **consumer always executes faster**, a buffer containing one location (or a small number of locations) is enough.
- If the **producer executes faster on average** and the program does not ask the producer to wait, only a buffer with an “infinite” number of locations would absorb the extra production.



If they **execute at about the same average speed**, a circular buffer helps to smooth the effects of occasional speeding up or slowing down in either thread’s execution and reduces waiting times, improving performance.



The key to a **circular buffer** is optimizing its size to minimize thread wait times while not wasting memory. If we determine that the producer often produces as many as three more values than the consumer can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer.

Implementing a Circular Buffer

Figures 17.9—17.10 demonstrate concurrent producer and consumer threads accessing a **circular buffer with synchronization** (Fig. 17.9). We implement the circular buffer as an array of three `int` elements. The consumer consumes a value only when the **array is not empty**, and the producer produces a value only when the **array is not full**. Again, the output statements used in this class’s critical sections are for demonstration purposes only.

```

1 // Fig. 17.9: CircularBuffer.h
2 // Synchronizing access to a shared three-element circular buffer.
3 #pragma once
4 #include <array>
5 #include <condition_variable>
6 #include <format>
7 #include <mutex>
8 #include <iostream>
9 #include <string>
10 #include <string_view>
```

Fig. 17.9 | Synchronizing access to a shared three-element circular buffer. (Part I of 3.)

```
11  using namespace std::string_literals;
12
13  class CircularBuffer {
14  public:
15      // place value into m_buffer
16      void put(int value) {
17          // critical section that requires a lock to modify shared data
18          {
19              // lock on m_mutex to be able to write to m_buffer
20              std::unique_lock dataLock{m_mutex};
21
22              // if no empty locations, wait on condition variable m_cv
23              if (m_occupiedCells == m_buffer.size()) {
24                  std::cout << "Buffer is full. Producer waits.\n\n";
25
26                  // wait on the condition variable; the lambda argument
27                  // ensures that if the thread gets the processor before
28                  // the m_buffer has open cells, the thread continues waiting
29                  m_cv.wait(dataLock,
30                          [&] {return m_occupiedCells < m_buffer.size();});
31
32          }
33
34          m_buffer[m_writeIndex] = value; // write to m_buffer
35          ++m_occupiedCells; // one more m_buffer cell is occupied
36          m_writeIndex = (m_writeIndex + 1) % m_buffer.size();
37          displayState(std::format("Producer writes {}", value));
38      } // dataLock's destructor releases the lock on m_mutex here
39
40      m_cv.notify_one(); // notify threads waiting to read from m_buffer
41  }
42
43  // return value from m_buffer
44  int get() {
45      int readValue; // will temporarily hold the next value read
46
47      // critical section that requires a lock to modify shared data
48      {
49          // lock on m_mutex to be able to write to m_buffer
50          std::unique_lock dataLock{m_mutex};
51
52          // if no data to read, place thread in waiting state
53          if (m_occupiedCells == 0) {
54              std::cout << "Buffer is empty. Consumer waits.\n\n";
55
56              // wait on the condition variable; the lambda argument
57              // ensures that if the thread gets the processor before
58              // there is data in the m_buffer, the thread continues waiting
59              m_cv.wait(dataLock, [&]() {return m_occupiedCells > 0;});
60          }
61
62          readValue = m_buffer[m_readIndex]; // read value from m_buffer
```

Fig. 17.9 | Synchronizing access to a shared three-element circular buffer. (Part 2 of 3.)

```

63         m_readIndex = (m_readIndex + 1) % m_buffer.size();
64         --m_occupiedCells; // one fewer m_buffer cells is occupied
65         displayState(std::format("Consumer reads {}", readValue));
66     } // dataLock's destructor releases the lock on m_mutex here
67
68     m_cv.notify_one(); // notify threads waiting to write to m_buffer
69     return readValue;
70 }
71
72 // display current operation and m_buffer state
73 void displayState(std::string_view operation) const {
74     std::string s;
75
76     // add operation argument and number of occupied m_buffer cells
77     s += std::format("{} (buffer cells occupied: {})\n{:<15}",
78                      operation, m_occupiedCells, "buffer cells:");
79
80     // add values in m_buffer
81     for (int value : m_buffer) {
82         s += std::format(" {:2d} ", value);
83     }
84
85     s += std::format("\n{:<15}", ""); // padding
86
87     // add underlines
88     for (int i{0}; i < m_buffer.size(); ++i) {
89         s += "---- "s;
90     }
91
92     s += std::format("\n{:<15}", ""); // padding
93
94     for (int i{0}; i < m_buffer.size(); ++i) {
95         s += std::format(" {}{} ",
96                           (i == m_writeIndex ? 'W' : ' '),
97                           (i == m_readIndex ? 'R' : ' '));
98     }
99
100    s += "\n\n";
101    std::cout << s; // display the state string
102 }
103 private:
104     std::condition_variable m_cv;
105     std::mutex m_mutex;
106
107     std::array<int, 3> m_buffer{-1, -1, -1}; // shared m_buffer
108     int m_occupiedCells{0}; // count number of buffers used
109     int m_writeIndex{0}; // index of next element to write to
110     int m_readIndex{0}; // index of next element to read
111 };

```

Fig. 17.9 | Synchronizing access to a shared three-element circular buffer. (Part 3 of 3.)

Data Members

Lines 104–110 define the class's data members:

- Lines 104–105 define the `std::condition_variable` `m_cv` and the `std::mutex` `m_mutex` for synchronizing access to `CircularBuffer`'s other data members.
- Line 107 creates and initializes the three-element `int` array `m_buffer`, representing the circular buffer.
- Variable `m_occupiedCells` (line 108) counts the number of `m_buffer` elements that contain readable data. When `m_occupiedCells` is 0, the circular buffer is empty, and the consumer must wait. When `m_occupiedCells` is 3 (the buffer's size), the circular buffer is full, and the producer must wait.
- Variable `m_writeIndex` (line 109) indicates the next location in which a value can be placed by a producer.
- Variable `m_readIndex` (line 110) indicates the position from which the next value can be read by a consumer.

Data members `m_buffer`, `m_occupiedCells`, `m_writeIndex` and `m_readIndex` are all part of the class's shared mutable data, so access to these variables must be synchronized to ensure that a `CircularBuffer` is thread-safe.

`CircularBuffer` Function `put`

The `put` function (lines 17–41) performs the same tasks as in Fig. 17.7, with a few modifications. Lines 24–32 check whether the `CircularBuffer` is full. If so, the producer must wait, so line 25 indicates that the Producer is waiting to perform its task. Then, lines 30–31 invoke `m_cv`'s `wait` function, causing the producer thread to release `m_mutex`'s lock and wait until there's space in the buffer to write a new value.

When execution continues at line 34, the producer places `value` in the circular buffer at location `m_writeIndex`. Line 35 increments `occupiedCells`, because the buffer contains a value the consumer can read. Then line 36 updates `m_writeIndex` for the producer's next put call. This line is the key to the buffer's circularity. When `writeIndex` increments past the end of the buffer, this line sets it back to 0. Next, line 37 calls `displayState` (lines 73–102) to display the value the producer wrote, the `occupiedCells` count, the cells' contents and the current `m_writeIndex` and `m_readIndex`. Reaching the end of the block at line 38 releases `m_mutex`'s lock. Line 40 calls `notify_one` on `m_cv` to transition a waiting thread to the ready state so that a waiting consumer thread (if there is one) can now try again to read a value from the buffer.

`CircularBuffer` Function `get`

The `get` function (lines 44–70) performs the same tasks as in Fig. 17.7, with a few minor modifications. Lines 53–60 (Fig. 17.9) determine whether all the buffer cells are empty, in which case the consumer must wait. If so, line 54 updates the output to indicate that the consumer is waiting to perform its task. Then line 59 invokes `m_cv`'s `wait` function, causing the consumer thread to release `m_mutex`'s lock and wait until data is available to read.

When execution continues at line 62, the local variable `readValue` is assigned the value at `m_buffer` location `m_readIndex`. Then line 63 updates `m_readIndex` for the next call to `CircularBuffer` function `get`. This line and line 36 implement the buffer's cir-

cularity. Line 64 decrements `m_occupiedCells`, because there's now an open buffer position in which the producer can place a value. Line 65 updates the output with the value being consumed, the `occupiedCells` count, the cells' contents and the current `m_writeIndex` and `m_readIndex`. Reaching line 66 releases `m_mutex`'s lock. Line 68 calls `m_cv`'s `notify_one` function to allow a waiting producer thread to attempt to write again. Then line 69 returns the consumed value to the caller.

CircularBuffer Function `displayState`

Function `displayState` (lines 73–102) builds then outputs a `string` containing the application's state. Lines 81–83 add the buffer cells' values to the string, using a "`:2d`" format specifier to format the contents of each cell with a leading space if it's a single digit. Lines 94–98 add to the `string` the current `m_writeIndex` and `m_readIndex` with the letters W and R, respectively. Note that we call this function only from the critical sections to ensure thread safety. Again, we should not do I/O in critical sections—we do this simply to produce useful outputs for demonstration purposes.

Testing Class `CircularBuffer`

Figure 17.10 contains the `main` function that launches the application. Line 13 creates a `CircularBuffer` object named `buffer`, and line 14 displays `buffer`'s initial state. Lines 17–37 and 40–60 create the `produce` and `consume` lambdas that the concurrent producer and consumer threads will execute. Lines 62–63 create the `std::jtheads` that call the `produce` and `consume` lambdas. When `main` ends, the `std::jtheads` go out of scope, automatically joining the threads.

```

1 // Fig. 17.10: SharedBufferTest.cpp
2 // Concurrent threads manipulating a synchronized circular buffer.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <mutex>
7 #include <random>
8 #include <thread>
9 #include "CircularBuffer.h"
10
11 int main() {
12     // Create CircularBuffer to store ints and display initial state
13     CircularBuffer buffer;
14     buffer.displayState("Initial State");
15
16     // Lambda expression that produces the values 1-10 and sums them
17     auto produce{
18         [&buffer](){
19             // Set up random-number generation
20             std::random_device rd;
21             std::default_random_engine engine{rd()};
22             std::uniform_int_distribution ints{0, 3000};
23
24             int sum{0};

```

Fig. 17.10 | Concurrent threads manipulating a synchronized circular buffer. (Part 1 of 4.)

```

25
26     for (int count{1}; count <= 10; ++count) {
27         // get random sleep time then sleep
28         std::chrono::milliseconds sleepTime{ints(engine)};
29         std::this_thread::sleep_for(sleepTime);
30
31         buffer.put(count); // set value in buffer
32         sum += count; // add count to sum of values produced
33     }
34
35     std::cout << "Producer done producing\nTerminating Producer\n\n";
36 }
37 };
38
39 // lambda expression that consumes the values 1-10 and sums them
40 auto consume{
41     [&buffer]() {
42         // set up random-number generation
43         std::random_device rd;
44         std::default_random_engine engine{rd()};
45         std::uniform_int_distribution ints{0, 3000};
46
47         int sum{0};
48
49         for (int count{1}; count <= 10; ++count) {
50             // get random sleep time then sleep
51             std::chrono::milliseconds sleepTime{ints(engine)};
52             std::this_thread::sleep_for(sleepTime);
53
54             sum += buffer.get(); // get buffer value and add to sum
55         }
56
57         std::cout << std::format("{} {}\n{}{}\n\n",
58             "Consumer read values totaling", sum, "Terminating Consumer");
59     }
60 };
61
62 std::jthread producer{produce}; // start producer thread
63 std::jthread consumer{consume}; // start consumer thread
64 }
```

Initial State (buffer cells occupied: 0)
 buffer cells: -1 -1 -1

 WR

Buffer is empty. Consumer waits.

Producer writes 1 (buffer cells occupied: 1)
 buffer cells: 1 -1 -1

 R W

(continued...)

Fig. 17.10 | Concurrent threads manipulating a synchronized circular buffer. (Part 2 of 4.)

```

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:   1   -1   -1
----- ----- -----
                           WR

Buffer is empty. Consumer waits.

Producer writes 2 (buffer cells occupied: 1)
buffer cells:   1   2   -1
----- ----- -----
                           R   W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:   1   2   -1
----- ----- -----
                           WR

Buffer is empty. Consumer waits.

Producer writes 3 (buffer cells occupied: 1)
buffer cells:   1   2   3
----- ----- -----
                           W       R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:   1   2   3
----- ----- -----
                           WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:   4   2   3
----- ----- -----
                           R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:   4   5   3
----- ----- -----
                           R       W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:   4   5   3
----- ----- -----
                           R   W

Consumer reads 5 (buffer cells occupied: 0)
buffer cells:   4   5   3
----- ----- -----
                           WR

Producer writes 6 (buffer cells occupied: 1)
buffer cells:   4   5   6
----- ----- -----
                           W       R

```

Fig. 17.10 | Concurrent threads manipulating a synchronized circular buffer. (Part 3 of 4.)

Producer writes 7 (buffer cells occupied: 2)
buffer cells: 7 5 6

W R

Producer writes 8 (buffer cells occupied: 3)
buffer cells: 7 8 6

WR

Buffer is full. Producer waits.

Consumer reads 6 (buffer cells occupied: 2)
buffer cells: 7 8 6

R W

Producer writes 9 (buffer cells occupied: 3)
buffer cells: 7 8 9

WR

Buffer is full. Producer waits.

Consumer reads 7 (buffer cells occupied: 2)
buffer cells: 7 8 9

W R

Producer writes 10 (buffer cells occupied: 3)
buffer cells: 10 8 9

WR

Producer done producing
Terminating Producer

Consumer reads 8 (buffer cells occupied: 2)
buffer cells: 10 8 9

W R

Consumer reads 9 (buffer cells occupied: 1)
buffer cells: 10 8 9

R W

Consumer reads 10 (buffer cells occupied: 0)
buffer cells: 10 8 9

WR

Consumer read values totaling 55
Terminating Consumer

Fig. 17.10 | Concurrent threads manipulating a synchronized circular buffer. (Part 4 of 4.)

Analyzing the Output

When the producer writes a value or the consumer reads a value, the program outputs a message indicating the action performed, `m_buffer`'s contents, and the `m_writeIndex ("W")` and `m_readIndex ("R")` locations. In the sample output, the consumer immediately waits because it tries to consume before the producer executes. Next, the producer writes 1. The buffer then contains 1 in the first cell and -1 (the default value we use for output purposes) in the other two cells. The write index is now at the second cell, while the read index is still at the first cell. Next, the consumer reads 1. The buffer contains the same values, but the read index is now at the second cell. The consumer then tries to read again, but the buffer is empty, and the consumer must wait. The consumer also waits after reading 2. Later in the output, the producer fills the buffer twice and subsequently waits each time.

17.8 Readers and Writers

Our producer-consumer examples used a single producer and a single consumer—common in many applications with threads that share mutable data. Some systems require multiple consumer threads (called readers) that read data and multiple producer threads (called writers) that write it. This is known as the **readers and writers problem**.

For example, there may be many more readers than writers in an airline reservation system. Many inquiries will be made against the database of available flight information before a customer actually purchases a particular seat on a particular flight. The key observation is that if you have multiple readers, they can read simultaneously without thread-safety issues because they do not modify the data. A writer still requires exclusive access to the critical section that modifies the data—there can be no other writers and no readers.

The C++ features that support multiple readers and writers are `std::shared_mutex`, `std::shared_lock` and `std::condition_variable_any`. A `std::shared_mutex`⁶³ (from the `<mutex>` header) allows one writer or multiple readers to own its lock:

- If the lock is available, a writer acquires it via a `lock_guard` or a `unique_lock`. Otherwise, the writer is **blocked** and must wait for the lock to become available. **While a writer holds a shared_mutex's lock, no other threads may acquire it.**
- If the lock is available, a reader acquires it via a `std::shared_lock`⁶⁴ (from the `<mutex>` header); otherwise, the reader is blocked and must wait for the lock to become available. While a reader holds a `shared_mutex`'s lock, only other readers may acquire the lock.

As in our synchronized producer-consumer examples, we must ensure that readers and writers perform their tasks only when it's their turn:

- If a reader is reading when a writer arrives, the writer must **wait** for the lock to become available. Also, any additional readers that subsequently arrive must **wait** until the currently waiting writer executes and notifies them that it has finished writing. The waiting writer would be **indefinitely postponed** if a stream of arriving readers were allowed to read.

63. “`std::shared_mutex`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/shared_mutex.

64. “`std::shared_lock`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/shared_lock.

- If a writer is writing when a reader arrives, the reader must `wait` for the lock to become available. Also, any additional writers that subsequently arrive must `wait` until the currently waiting readers read. The waiting readers would be `indefinitely postponed` if a stream of arriving writers were allowed to write.

Once again, we can use condition variables to manage these conditions—one for the readers and a separate one for the writers.

Readers use `shared_locks`, but class `condition_variable` requires `unique_locks`. So, for readers, use a `std::condition_variable_any`⁶⁵ object to manage waiting readers. A `condition_variable_any` works like a `condition_variable` but supports other lock types. When a writer finishes writing, it would `notify_all` currently waiting readers that it's time to read by calling the `condition_variable_any` object's `notify_all` function. All reader threads waiting for that condition would then transition to the `ready` state and become eligible to reacquire the lock.

Writer thread(s) would still use `unique_lock` for exclusive access to the shared mutable data, so we can manage waiting writers with a `condition_variable`. When all active readers finish reading, the program would call the `condition_variable` object's `notify_all` function. All the waiting writer threads would move to the `ready` state and become eligible to reacquire the lock. However, **only one writer** would reacquire the lock and proceed.

17.9 Cooperatively Cancelling jthreads

When a multithreaded application needs to terminate, it's good practice to shut down threads that are still performing tasks, so they can release the resources they're using.⁶⁶ For example, they might need to close files, database connections and network connections. Before C++20, there was no standard mechanism for threads to cooperate with one another to terminate gracefully. This is another defect of `thread`.

C++20 added `cooperative cancellation` to fix this problem, enabling programs to notify threads when it's time for them to terminate. The task executing in a thread can watch for such notifications, then complete critical work, release resources and terminate.

Figure 17.11 demonstrates `cooperative cancellation` between `threads` using features from the `<stop_token>` header. As you'll see, `jthread` integrates these features.

```

1 // Fig. 17.11: CooperativeCancellation.cpp
2 // Using a std::jthread's built-in stop_source
3 // to request that the std::jthread stop executing.
4 #include <chrono>
5 #include <format>
6 #include <iostream>
7 #include <mutex>
8 #include <random>

```

Fig. 17.11 | Using a `std::jthread`'s built-in `stop_source` to request that the `std::jthread` stop executing. (Part 1 of 3.)

65. “`std::condition_variable`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/condition_variable_any.

66. Anthony Williams, “Concurrency in C++20 and Beyond,” October 16, 2019. Accessed April 18, 2023. https://www.youtube.com/watch?v=jozHW_B3D4U.

```

9  #include <iostream>
10 #include <stop_token>
11 #include <string>
12 #include <string_view>
13 #include <thread>
14
15 // get current thread's ID as a string
16 std::string id() {
17     std::ostringstream out;
18     out << std::this_thread::get_id();
19     return out.str();
20 }
21
22 int main() {
23     // each printTask iterates until a stop is requested by another thread
24     auto printTask{
25         [&](std::stop_token token, std::string name) {
26             // set up random-number generation
27             std::random_device rd;
28             std::default_random_engine engine{rd()};
29             std::uniform_int_distribution ints{500, 1000};
30
31             // register a function to call when a stop is requested
32             std::stop_callback callback(token, [&](){
33                 std::cout << std::format(
34                     "{} told to stop by thread with id {}\n", name, id());
35             });
36
37             while (!token.stop_requested()) { // run until stop requested
38                 auto sleepTime{std::chrono::milliseconds{ints(engine)}};
39
40                 std::cout << std::format(
41                     "{} (id: {}) going to sleep for {} ms\n",
42                     name, id(), sleepTime.count());
43
44                 // put thread to sleep for sleepTime milliseconds
45                 std::this_thread::sleep_for(sleepTime);
46
47                 // show that task woke up
48                 std::cout << std::format("{} working.\n", name);
49             }
50
51             std::cout << std::format("{} terminating.\n", name);
52         }
53     };
54
55     std::cout << std::format("MAIN (id: {}) STARTING TASKS\n", id());
56
57     // create two jthreads that each call printTask with a string argument
58     std::jthread task1{printTask, "Task 1"};
59     std::jthread task2{printTask, "Task 2"};
60 }
```

Fig. 17.11 | Using a `std::jthread`'s built-in `stop_source` to request that the `std::jthread` stop executing. (Part 2 of 3.)

```

61  // put main thread to sleep for 2 seconds
62  std::cout << "\nMAIN GOING TO SLEEP FOR 2 SECONDS\n\n";
63  std::this_thread::sleep_for(std::chrono::seconds{2});
64
65  std::cout << std::format("\nMAIN (id: {}) ENDS\n\n", id());
66 }

```

```

MAIN (id: 16352) STARTING TASKS

MAIN GOING TO SLEEP FOR 2 SECONDS

Task 1 (id: 14048) going to sleep for 708 ms
Task 2 (id: 10504) going to sleep for 995 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 940 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 926 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 875 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 519 ms

MAIN (id: 16352) ENDS

Task 2 told to stop by thread with id 16352
Task 2 working.
Task 2 terminating.
Task 1 told to stop by thread with id 16352
Task 1 working.
Task 1 terminating.

```

Fig. 17.11 | Using a `std::jthread`'s built-in `stop_source` to request that the `std::jthread` stop executing. (Part 3 of 3.)

Each `jthread` internally maintains a `std::stop_source`, which has an associated `std::stop_token`. A `jthread`'s task function can optionally receive this token as its first parameter. The task function then can periodically call the token's `stop_requested` member function to determine whether the task should stop executing. When

- another thread calls the `jthread`'s `request_stop` member function, or
- the `jthread`'s destructor calls `request_stop` as the `jthread` goes out of scope,

the `stop_source` notifies its associated `stop_token` that a stop has been requested. Subsequent calls to the `stop_token`'s `stop_requested` member function will return `true`. The task can then gracefully terminate its execution. If the task function never calls `stop_requested`, the corresponding `jthread` continues executing—hence, the term **cooperative cancellation**.

In this example, we'll let the `jthread`'s destructor call the `request_stop` member function, enabling the program to terminate soon after `main` completes. We create two `jthreads` (lines 58–59) that call the `printTask` lambda (lines 24–53). Each `jthread` passes its `stop_token` to the lambda. Lines 37–49 in the lambda loop continuously

- printing the task name, thread ID and sleep time,

- sleeping, then
- printing the task name and saying that the lambda is performing work.

The loop executes until the corresponding `jthread`'s internal `stop_source` receives a call to its `request_stop` member function. In this program, that call occurs in the `jthread`'s destructor when the `jthreads` go out of scope at the end of `main`.

Optional `stop_callback`

Lines 32–35 also demonstrate that you can register an optional function to call when a stop is requested. The `std::stop_callback`⁶⁷ constructor receives as arguments a `stop_token` and a function with no parameters—in this case, a lambda. The constructor registers the function with the given `stop_token`. When the `stop_token` is notified that a stop was requested, it calls the function registered by the `stop_callback` on the thread that requested the stop (the `main` thread in this example). Any number of `stop_callbacks` can be created for a given `stop_token`. The order in which their functions execute is not specified. Our lambda simply displays a `string` to show that the callback was indeed called, but this could be used to perform cleanup operations before a thread terminates.⁶⁸

Analyzing the Output

Throughout the sample output, Task 1 and Task 2 sleep and work. Once `main` ends, the `jthreads` executing these tasks go out of scope, and their destructors call each `jthread`'s `request_stop` member function to notify the tasks that they should terminate. In our output after "MAIN (id: 16352) ENDS", you can see when each `jthread` received its `request_stop` call—our `stop_callback` displays the task name followed by a message that includes the ID of the thread that told the task to stop (the `main` thread in this example).⁶⁹ In each case, the corresponding task then finishes its work and terminates.

17.10 Launching Tasks with `std::async`

Figure 17.12 demonstrates `std::async`—a higher-level way to launch a task in a separate thread. In this example, we'll implement a potentially long-running task—determining whether a large integer is prime and, if not, determining its prime factors.



Security, Encryption and Prime Factorization for Enormous Integers

Security is a crucial application. Prime factorization^{70,71,72} is a vital aspect of the RSA Public-Key Cryptography algorithm, which is commonly used to secure data transmitted

67. "std::stop_callback." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/stop_callback.

68. Williams, "Concurrency in C++20 and Beyond."

69. If the stop is requested before the `stop_callback` is constructed, then the thread ID displayed will be that of the thread constructing the `stop_callback` (per https://en.cppreference.com/w/cpp/thread/stop_callback).

70. "Prime Factorization," July 2, 2020. Accessed April 18, 2023. <https://www.cuemath.com/numbers/prime-factorization/>.

71. Striver, "Prime Factors of a Big Number," May 28, 2021. Accessed April 18, 2023. <https://www.geeksforgeeks.org/prime-factors-big-number/>.

72. Ehud Shalit, "Prime Numbers—Why Are They So Exciting?" September 7, 2018. Accessed April 18, 2023. <https://kids.frontiersin.org/articles/10.3389/frym.2018.00040>.

over the Internet.^{73,74,75} Industrial-strength RSA works with enormous prime numbers consisting of hundreds of digits. The sheer amount of time required to factor the product of those primes—even for the most powerful supercomputers in use today—is a key reason why RSA is so secure. Public-key cryptography also is used to secure the blockchain technology behind cryptocurrencies like Bitcoin.⁷⁶ The RSA Public-Key Cryptography case-study exercise at the end of Chapter 9 presents the RSA algorithm.

Overview of This Example

This example’s task to execute is defined by the `getFactors` function (lines 27–68), which determines whether a number is prime⁷⁷ and, if not, determines its prime factors. Each thread in this program executes until `getFactors` returns its result—a `std::tuple` containing the task name, the number `getFactors` processed, a `bool` indicating whether the number is prime and a `vector` of the number’s factors. To ensure that our tasks run for at least a few seconds each, we used two 19-digit numbers, including a prime value from the University of Tennessee Martin’s Prime Pages website.⁷⁸ Its prime-number research includes lists of prime values by their numbers of digits.

Figure 17.12 consists of the following functions:

- Function `id` (lines 13–17) converts a `unique std::thread::id` to a `std::string`.
- Function `getFactors` (lines 27–68) attempts to find an integer’s factors.
- Function `proveFactors` (lines 71–90) confirms that a given set of prime factors, when multiplied together, reproduces a corresponding non-prime value.
- Function `displayResults` (lines 93–117) displays a task’s results.
- Function `main` (lines 119–133) launches the `getFactors` tasks, waits for them to complete, then displays their results.

We’ve split this program into pieces for discussion purposes. After the program, we show sample outputs. In Fig. 17.12, lines 3–10 `#include` the headers used in this program.

73. “RSA (Cryptosystem).” Accessed April 18, 2023. [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)).

74. “RSA Algorithm.” Accessed April 18, 2023. https://simple.wikipedia.org/wiki/RSA_algorithm.

75. K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch, “PKCS #1: RSA Cryptography Specifications Version 2.2,” November 2016. Accessed April 18, 2023. <https://tools.ietf.org/html/rfc8017>.

76. Sarah Rothrie, “How Blockchain Cryptography Is Fighting the Rise of Quantum Machines,” December 28, 2018. Accessed April 18, 2023. <https://coinalcentral.com/blockchain-cryptography-quantum-machines/>.

77. “Primality Test.” Accessed April 18, 2023. https://en.wikipedia.org/wiki/Primality_test.

78. “Index: Numbers.” Accessed April 18, 2023. <https://primes.utm.edu/curios/index.php>.

```

1 // Fig. 17.12: async.cpp
2 // Prime-factorization tasks performed in separate threads
3 #include <cmath>
4 #include <format>
5 #include <future> // std::async
6 #include <iostream>
7 #include <sstream>
8 #include <string>
9 #include <tuple>
10 #include <vector>
11
12 // get current thread's ID as a string
13 std::string id() {
14     std::ostringstream out;
15     out << std::this_thread::get_id();
16     return out.str();
17 }
18

```

Fig. 17.12 | Prime-factorization tasks launched with `std::async`.

Type Aliases

Lines 20 and 24 define type aliases we use to simplify type declarations:

- Factors (line 20) is an alias for a vector of pairs, each containing a prime factor and how many times an integer was divisible by that factor. For example, 8 has three factors of 2.
- FactorResults (line 24) is the `getFactors` function’s return type. This alias represents a tuple that contains a task’s name (`std::string`), the number for which we’ll determine the prime factors (`long long`), whether the number is prime (`bool`) and the prime factors (`Factors`).

```

19 // type alias for vector of factor/count pairs
20 using Factors = std::vector<std::pair<long long, int>>;
21
22 // type alias for a tuple containing a task name,
23 // a number, whether the number is prime and its factors
24 using FactorResults = std::tuple<std::string, long long, bool, Factors>;
25

```

getFactors Function to Determine Prime Factorization of an Integer

This program launches separate threads that each call function `getFactors` (lines 27–68) to find an integer’s factors or determine it is prime. The function receives a task name to display in outputs and the number to factor and returns a `FactorResults` object.

```

26 // performs prime factorization
27 FactorResults getFactors(std::string name, long long number) {
28     std::cout << std::format(
29         "{}: Thread {} executing getFactors for {}\n", name, id(), number);
30

```

```

31 long long originalNumber{number}; // copy to place in FactorResults
32 Factors factors; // vector of factor/count pairs
33
34 // lambda that divides number by a factor and stores factor/count
35 auto factorCount{
36     [&](int factor) {
37         int count{0}; // how many times number is divisible by factor
38
39         // count how many times number is divisible by factor
40         while (number % factor == 0) {
41             ++count;
42             number /= factor;
43         }
44
45         // store pair containing the factor and its count
46         if (count > 0) {
47             factors.push_back({factor, count});
48         }
49     };
50 };
51
52 factorCount(2); // count how many times number is divisible by 2
53
54 // number is now odd; store each factor and its count
55 for (int i{3}; i <= std::sqrt(number); i += 2) {
56     factorCount(i); // count how many times number is divisible by i
57 }
58
59 // add last prime factor
60 if (number > 2) {
61     factors.push_back({number, 1});
62 }
63
64 bool isPrime{factors.size() == 1 && get<int>(factors[0]) == 1};
65
66 // initialize the FactorResults object returned by getFactors
67 return {name, originalNumber, isPrime, factors};
68 }
69

```

The function operates as follows:

- Lines 28–29 display the executing task’s name and the ID of the thread in which `getFactors` is executing.
- If `number` has prime factors, this algorithm will modify `number`, so line 31 copies `number` for inclusion in the `FactorResults`.
- Line 32 defines the `Factors` object—a vector of factor/count pairs.
- Lines 35–50 define the lambda `factorCount`, which counts how many times `number` (which is captured by reference) is divisible by the lambda’s `factor` argument. While `number` is divisible by `factor` (line 40), we increment the count and divide `number` by `factor`. Then, if the count is greater than 0, line 47 adds a new factor/count pair to the `factors` object.

- Line 52 calls `factorCount` for the factor 2. Then, lines 55–57 repeatedly call it for the odd values 3 and above while `i` is less than or equal to `number`'s square root.
- Lines 60–62 check whether `number` is still greater than 2. If so, this is the last factor to add to `factors`.
- If `number` is prime, `factors` will contain only the number itself, and its factor count will be 1. Line 64 checks this and sets the `bool` variable `isPrime` accordingly.
- Finally, line 67 initializes the `FactorResults` tuple that `getFactors` returns, using the task name, `originalNumber`, `isPrime` and `factors`.

proveFactors Function to Confirm Prime Factorization

Lines 71–90 define `proveFactors`, which confirms that a non-prime integer value can be reproduced by calculating the product of its prime factors:

- Line 72 initializes `proof` to 1.
- For each prime factor/count pair in `factors` (line 76), lines 77–79 iterate `count` times, multiplying `proof` by that factor.
- Lines 83–89 check if `number` matches `proof` and display an appropriate message.

```

70 // multiply the factors and confirm they reproduce number
71 void proveFactors(long long number, const Factors& factors) {
72     long long proof{1};
73
74     // for each factor/count pair, unpack it then multiply proof
75     // by factor the number of times specified by count
76     for (const auto& [factor, count] : factors) {
77         for (int i{0}; i < count; ++i) {
78             proof *= factor;
79         }
80     }
81
82     // confirm proof and number are equal
83     if (proof == number) {
84         std::cout << std::format(
85             "\nProduct of factors matches original value ({})\n", proof);
86     }
87     else {
88         std::cout << std::format("\n{} != {}{}\n", proof, number);
89     }
90 }
91

```

displayResults Function to Display Prime Factorization

Lines 93–117 define `displayResults`, which `main` calls to display each `FactorResults` tuple received from this program's tasks:

- Line 96 unpacks the tuple into variables `name` (`std::string`), `number` (`long long`), `isPrime` (`bool`) and `factors` (`Factors`). Each variable's type is inferred from the tuple's type (line 24).
- Line 98 displays the task name.

- If the number is prime (line 101), line 102 displays a message. Otherwise, lines 105–110 display the non-prime number's prime factors and their counts.
- Finally, if the number is not prime, line 115 calls `proveFactors` to check whether the prime factors, when multiplied, reproduce the number.

```
92 // show a task's FactorResults
93 void displayResults(const FactorResults& results) {
94     // unpack results into name (std::string), number (long long),
95     // isPrime (bool) and factors (Factors)
96     const auto& [name, number, isPrime, factors] {results};
97
98     std::cout << std::format("\n{} results:\n", name);
99
100    // display whether value is prime
101    if (isPrime) {
102        std::cout << std::format("{} is prime\n", number);
103    }
104    else { // display prime factors
105        std::cout << std::format("{}'s prime factors:\n\n", number);
106        std::cout << std::format("{:<12}{:<8}\n", "Factor", "Count");
107
108        for (const auto& [factor, count] : factors) {
109            std::cout << std::format("{:<12}{:<8}\n", factor, count);
110        }
111    }
112
113    // if not prime, prove that factors produce the original number
114    if (!isPrime) {
115        proveFactors(number, factors);
116    }
117 }
118 }
```

Creating and Executing a Task: Function template `std::async`

The `<future>` header (line 5) contains features that enable you to execute asynchronous tasks and receive the results of those tasks when they finish executing.

```
119 int main() {
120     std::cout << "MAIN LAUNCHING TASKS\n";
121     auto future1{std::async(std::launch::async,
122                           getFactors, "Task 1", 1016669006116682993)}; // not prime
123     auto future2{std::async(std::launch::async,
124                           getFactors, "Task 2", 1000000000000000003)}; // prime
125
126     std::cout << "\nWAITING FOR TASK RESULTS\n";
127
128     // wait for results from each task, then display the results
129     displayResults(future1.get());
130     displayResults(future2.get());
131
132     std::cout << "\nMAIN ENDS\n";
133 }
```

MAIN LAUNCHING TASKS

WAITING FOR TASK RESULTS

```
Task 1: Thread 5032 executing getFactors for 1016669006116682993
Task 2: Thread 2952 executing getFactors for 10000000000000000003
```

Task 1 results:

```
1016669006116682993's prime factors:
```

Factor	Count
1000000007	1
1016668999	1

```
Product of factors matches original value (1016669006116682993)
```

Task 2 results:

```
10000000000000000003 is prime
```

```
MAIN ENDS
```

Lines 121–122 and 123–124 use the `std::async` function template⁷⁹ to create two threads that execute `getFactors` asynchronously. Function `std::async` has two versions. The one used here takes three arguments:

- The launch policy (from the `std::launch` enum) is `std::launch::async`, `std::launch::deferred` or both separated by a bitwise OR (`|`) operator. The value `std::launch::async` indicates that the function specified in the second argument should execute in a separate thread, whereas `std::launch::deferred` indicates that it should execute in the same thread. Combining these values lets the system choose whether to execute asynchronously or synchronously.
- The task to execute is specified by a function pointer, function object or lambda.
- Any remaining arguments are passed by `std::async` to the task function. We passed a string name for the task and a long `long` value specifying the number for which to calculate the prime factorization.

The other `std::async` version does not receive a launch policy argument—it chooses the launch policy for you.

If `std::async` receives the launch policy `std::launch::async` but cannot create the thread, it throws a `std::system_error` exception. If `std::async` creates the thread successfully, the task function begins executing in the new thread.

`std::future`, `std::promise` and Inter-Thread Communication

Function `std::async` returns an object of class template `std::future`, which enables **inter-thread communication** between the thread that calls `async` and the task `async` executes. The C++ Core Guidelines recommend using a `future` to return a result from an asynchronous task.⁸⁰

79. “`std::async`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/thread/async>.

80. C++ Core Guidelines, “CP.60: Use a `future` to Return a Value from a Concurrent Task.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-future>.

“Under the hood,” `async` uses a `std::promise` object from which it obtains the returned `future`. When the task completes, `async` stores the task’s result in the `promise`. `async`’s caller uses the `future` to access the result in the `promise`—in our case, a `FactorResults` object. You do not need to work with the `promise` directly.

To ensure that the program does not terminate until the tasks complete and to receive the results from each task, lines 129–130 call each `future`’s `get` member function. If the task is still running, `get` blocks the calling thread, which waits until the task completes; otherwise, `get` immediately returns the task’s result. Function `get`’s return type is whatever the task function returns. Once the results are available, lines 129–130 pass them to function `displayResults`. The program’s output shows unique thread IDs, confirming that both `getFactors` tasks executed in separate threads.

If `async`’s task throws an exception, the `future` contains the exception rather than the task’s result, and calling `get` rethrows the exception. If multiple threads need access to an asynchronous task’s result, you can use `std::shared_future` objects.⁸¹

Err

`std::packaged_task` vs. `std::async`

Another way to launch asynchronous tasks is `std::packaged_task`.⁸² The key differences between `async` and `packaged_task` are as follows:

- You must call its `get_future` member function to get the associated `future` object for obtaining the task’s results at a later time.
- A `packaged_task` executes on the thread that calls the task’s `operator()` function.
- To execute a `packaged_task` in a separate thread, you create the thread and initialize it with `std::move(yourPackagedTaskObject)`. When the thread completes, you can call `get` on the task’s `future` object to obtain the result.

17.11 Thread-Safe, One-Time Initialization

Sometimes a variable should be initialized exactly once, even when concurrent threads try to execute the variable’s initialization statement. Here, we discuss two mechanisms for one-time, thread-safe initialization of a variable.

static Local Variables

When a function defines a `static` local variable and concurrent threads execute that function, the C++ standard specifies that the variable’s initialization “is performed the first time control passes through its declaration.”^{83,84} So, the first thread that executes the `static` local variable’s declaration performs the initialization. All other threads attempting to execute the `static` variable’s declaration are `blocked` until the first thread completes the vari-

81. “`std::shared_future`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/shared_future.

82. “`std::packaged_task`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/packaged_task.

83. C++ Standard, “8.8 Declaration Statement.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/stmt.dcl#4>.

84. “Storage Class Specifiers—static Local Variables.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/storage_duration#Static_local_variables.

able's initialization. Then the other threads calling the function skip the declaration and use the already initialized `static` local variable.

`once_flag` and `call_once`

Class `std::once_flag`⁸⁵ and function `std::call_once`⁸⁶ (from the `<mutex>` header) are used together to ensure that concurrent threads execute a function, lambda or function object exactly once. First, declare a `once_flag` object:

```
std::once_flag myFlag;
```

Then, use `call_once` to call your function:

```
std::call_once(myFlag, myFunction, arguments);
```

The first thread that executes the preceding statement will call *myFunction*, passing the specified *arguments* (if any). When a subsequent thread executes this statement, `call_once` returns immediately and does not call *myFunction*. For a common `call_once` use-case, see Arthur O'Dwyer's "Back to Basics: Concurrency" video.⁸⁷

17.12 A Brief Introduction to Atomics

The examples in Sections 17.6–17.7 ensured exclusive access to shared mutable data using the synchronization primitives `std::mutex`, `locks` and `std::condition_variable`. **Atomic types**⁸⁸ (from the `<atomic>` header⁸⁹) provide operations that cannot be divided into smaller steps, enabling threads to conveniently share mutable data without explicit synchronization and locking.

In a sense, atomics are higher level than `mutexes`, `locks` and `condition_variables`. However, atomic types and their operations are primarily considered to be low-level features intended to help library developers implement higher-level concurrency capabilities. For example, the Visual C++, GNU C++ and Clang C++ standard libraries each use atomics "under the hood" to implement C++20's `std::latch`, `std::barrier` and `std::semaphore` thread-coordination primitives (Sections 17.13–17.14).^{90,91,92}

We present simple examples of atomics. Most developers will prefer higher-level primitives, which help them quickly build correct, robust, efficient concurrent applica-

-
- 85. "std::once_flag." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/once_flag.
 - 86. "std::call_once." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/call_once.
 - 87. Arthur O'Dwyer, "Back to Basics: Concurrency," October 6, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=F6Ip7gCoSY>.
 - 88. Hans-J. Boehm and Lawrence Crowl, "C++ Atomic Types and Operations," October 3, 2007. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html>.
 - 89. "Atomic Operations Library." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/atomic>.
 - 90. "Microsoft's C++ Standard Library." Accessed April 18, 2023. <https://github.com/microsoft/STL>.
 - 91. "libstdc++." Accessed April 18, 2023. <https://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/files.html>.
 - 92. "libc++ Documentation." Accessed April 18, 2023. <https://github.com/llvm/llvm-project/tree/main/libcxx/>.

tions. For an extensive discussion of atomics' performance vs. that of related technologies, see the blog post, "A Concurrency Cost Hierarchy."⁹³

Incrementing an int, a `std::atomic<int>` and a `std::atomic_ref<int>` from Two Concurrent Threads

Figure 17.13 presents a simple demonstration of two concurrent threads incrementing

- an unprotected `int`,
- an object of type `std::atomic<int>` and
- an object of the class template `std::atomic_ref<int>`.

Class template `std::atomic_ref` was introduced in C++20 to enable atomic operations on a referenced variable (an `int` in this program). First, we'll show `std::atomic<int>` and `std::atomic_ref<int>` in action, then we'll say a bit more about atomics.

```

1 // Fig. 17.13: atomic.cpp
2 // Incrementing integers from concurrent threads
3 // with and without atomics.
4 #include <atomic>
5 #include <format>
6 #include <iostream>
7 #include <thread>
8
9 int main() {
10     int count1{0};
11     std::atomic<int> atomicCount{0};
12     int count2{0};
13     std::atomic_ref<int> atomicRefCount{count2};
14
15     {
16         std::cout << "Two concurrent threads incrementing int count1, "
17             << "atomicCount and atomicRefCount\n\n";
18
19         // lambda to increment counters
20         auto incrementer{
21             [&]() {
22                 for (int i{0}; i < 1000; ++i) {
23                     ++count1; // no synchronization
24                     ++atomicCount; // ++ is an atomic operation
25                     ++atomicRefCount; // ++ is an atomic operation
26                     std::this_thread::yield(); // force thread to give up CPU
27                 }
28             }
29         };
30
31         std::jthread t1{incrementer};
32         std::jthread t2{incrementer};
33     }

```

Fig. 17.13 | Incrementing integers from concurrent threads with and without atomics. (Part I of 2.)

93. Travis Downs, "A Concurrency Cost Hierarchy," July 6, 2020. Accessed April 18, 2023. <https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html>.

```

34
35     std::cout << std::format("Final count1: {}\n", count1);
36     std::cout << std::format("Final atomicCount: {}\n", atomicCount.load());
37     std::cout << std::format("Final count2: {}\n", count2);
38 }

```

Two concurrent threads incrementing int count1, atomicCount and atomicRefCount

```

Final count1: 2000
Final atomicCount: 2000
Final count2: 2000

```

Two concurrent threads incrementing int count1, atomicCount and atomicRefCount

```

Final count1: 1554
Final atomicCount: 2000
Final count2: 2000

```

Fig. 17.13 | Incrementing integers from concurrent threads with and without atomics. (Part 2 of 2.)

Defining the Counters

Lines 10–13 define

- int variable count1,
- **std::atomic<int>** variable atomicCount,
- int variable count2 and
- **std::atomic_ref<int>** variable atomicRefCount.

The first three variables are initialized to 0, and **atomicRefCount** is initialized with a reference to the variable count2. We'll use atomicRefCount to show that you can indirectly manipulate count2 atomically through a reference.

Incrementing the Counters

Lines 15–33 use two **unsynchronized concurrent threads** to increment the counters 1,000 times per thread. If every increment works correctly, each counter's total should be 2,000 when these threads complete. Lines 20–29 define the lambda **incrementer**, which uses the operator **++** to increment each of count1 (line 23), atomicCount (line 24) and atomicRefCount (line 25). **Operator ++ is one of a limited number of operations you can perform on atomic objects. When one thread is executing ++ on an atomic object, the increment is guaranteed to complete before another thread can modify or view that atomic object's value.** The same is not guaranteed for a non-atomic int variable. For a complete list of atomic operations, see

<https://en.cppreference.com/w/cpp/atomic/atomic>

At today's processor speeds, a thread can perform loop iterations quickly, so multiple threads incrementing an int without synchronization might appear to work correctly, as shown in this program's first output. For this reason, line 26 uses the **std::this_thread** namespace's **yield function** to force the currently executing thread to relinquish the pro-



cessor after each loop iteration. Like sleeping, yielding helps us emphasize that you cannot predict the relative speeds of asynchronous concurrent threads.

Lines 31–32 create `std::jthreads` that each execute `incrementer`. Then, the block terminates, and the `jthreads` go out of scope, automatically joining the threads. Finally, lines 35–37 display the values of `count1`, `atomicCount` and `count2`—`atomicCount`'s `Load` function obtains the object's integer value. The variable `count2` was incremented indirectly via `atomicRefCount`.

Analyzing the Sample Outputs

Lines 20–29 should increment the `count1`, `atomicCount` and `count2` (indirectly through `atomicRefCount`) 2,000 times. However, the sample outputs show that incrementing the `int count` from concurrent threads without synchronization can produce different totals each time, including the seemingly correct total in the first output. Incrementing `atomicCount` and incrementing `count2` indirectly through `atomicRefCount` from concurrent threads without synchronization is guaranteed to produce the proper total of 2,000 for each.



More About Atomic Types^{94,95}

There are predefined `std::atomic` class template specializations and corresponding type aliases for type `bool` and every integral type. You also can specialize `std::atomic` for

- pointer types,
- floating-point types and
- trivially copyable types⁹⁶—that is, the compiler provides the copy and move constructors, copy and move assignment operators, and destructor, and there are no virtual functions.

The specializations for integral, pointer and, as of C++20, floating-point types each support adding a value to or subtracting a value from an atomic data item. The integral specializations also provide atomic bitwise `&=` (and), `|=` (or) and `^=` (xor) operations.

C++20 Atomic Smart Pointers and Atomic Pointers^{97,98}

C++20 adds atomic smart pointer specializations for `std::atomic<shared_ptr<T>>` and `std::atomic<weak_ptr<T>>`, enabling atomic pointer manipulations of these smart pointer types.

For all atomic pointers, only the pointer manipulations are atomic—for example,

- aiming the pointer at a different object,
- incrementing the pointer by an integer or
- decrementing the pointer by an integer.

Atomic pointers can be used to implement thread-safe, linked data structures.

-
94. C++ Standard, “Atomic Operations Library.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/atomics>.
 95. “`std::atomic`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/atomic/atomic>.
 96. C++ Standard, “Properties of Classes.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/class.prop>.
 97. C++ Standard, “Partial Specializations for Smart Pointers.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/util.smartptr.atomic>.
 98. “`std::atomic`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/atomic/atomic>.

std::atomic_ref Class Template

Like `std::atomic`, the `std::atomic_ref` class template can be specialized for any primitive type, pointer type or trivially copyable type. However, a `std::atomic_ref` object is initialized with a reference rather than a value. You use the `atomic_ref` to perform atomic operations on the referenced object. Multiple `atomic_refs` can refer to the same object. The referenced object must outlive the `std::atomic_ref`; otherwise, a dangling reference or pointer will result.^{99,100}



17.13 Coordinating Threads with C++20 Latches and Barriers

So far, we've coordinated threads using `std::mutex`, `std::condition_variable` and locks. C++20 provides higher-level thread-coordination types `std::latch` and `std::barrier`,¹⁰¹ which do not require explicit use of mutexes, condition variables and locks.¹⁰²



17.13.1 C++20 std::latch

A `std::latch` (from the `<latch>` header) is a single-use gateway in your code that remains closed until a specified number of threads reach the latch. At that point, the gateway remains open permanently.¹⁰³ The gateway serves as a one-time synchronization point, allowing threads to wait until a specified number of threads reach that point. Latches are simpler to use than mutexes and condition variables for coordinating threads.

Consider a parallel sorting algorithm that

- launches several worker threads to sort portions of a large array,
- waits for the worker threads to complete, then
- merges the sorted sub-arrays into the final sorted array.

Assume the algorithm uses two worker threads, each sorting half the array. The algorithm can use a `std::latch` object to wait until the workers are done:

- First, the algorithm creates a `std::latch` with a non-zero count. In this case, it needs two worker threads to complete, so we initialize the `latch` to 2.
- Each worker has a reference to the same `latch`.
- After launching the workers, the algorithm waits on that `latch`, which blocks the algorithm from continuing.
- When a thread reaches the `latch`, the thread reduces the `latch`'s count by 1 (known as **signaling the latch**) to indicate that it finished sorting a sub-array.

99. C++ Standard, “31.7 Class Template `atomic_ref`.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/atomics.ref.generic>.

100. “`std::atomic_ref`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/atomic/atomic_ref.

101. C++ Standard, “Thread Support Library—Coordination Types,” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/thread.coord>.

102. Bryce Adelstein Lelbach, “The C++20 Synchronization Library,” October 24, 2019. November 30, 2021. <https://www.youtube.com/watch?v=Zcqwb3Cwqs4>.

103. C++ Standard, “Thread Support Library—Coordination Types—Latches.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/thread.latch>.

- When the **latch**'s count becomes 0, the **latch** permanently opens, unblocking the algorithm's thread so it can merge the results of its worker threads.

Attempting to Wait When the Gateway Is Permanently Open

Any number of threads can attempt to wait on a given **latch**. However, once the **latch** is open, other threads that attempt to wait on it simply pass through the gateway and continue executing.

Demonstrating Latches

Figure 17.14 uses a lambda named **task** (lines 22–34) to perform work that must complete before **main** is allowed to continue executing. As we've done throughout this chapter, we simulate the work by sleeping. The program operates as follows:

- Line 19 creates the **std::latch** **mainLatch** with the count 3. Any thread(s) waiting on **mainLatch** cannot continue executing until three threads reach **mainLatch**.
- Lines 40–46 launch three **jthreads**—each executes the **task** lambda.
- After **main** launches the **jthreads**, line 49 calls the **mainLatch**'s **wait member function**. If **mainLatch**'s count is greater than 0, **main** waits at line 49 until the latch's count reaches 0.
- When each **jthread**'s task call reaches line 32, it calls **mainLatch**'s **count_down member function** to signal the latch, decrementing its count. Though the task terminates in this program, it could continue working. The key is that the work **main** is waiting for should be completed by each thread before it **signals the latch**. In the outputs, note that the threads signal the latch in different orders.
- When **mainLatch**'s count reaches 0, any thread(s) waiting on **mainLatch** unblock and continue executing. In this program, **main** continues executing at line 51.
- To show that a latch is a **single-use gateway**, line 53 calls **mainLatch**'s **wait** function again. Since **mainLatch** is already open, **main** simply continues executing at line 54, displays another message then terminates.

```

1 // Fig. 17.14: LatchDemo.cpp
2 // Coordinate threads with a std::latch object.
3 #include <chrono>
4 #include <format>
5 #include <iostream>
6 #include <latch>
7 #include <random>
8 #include <string_view>
9 #include <thread>
10 #include <vector>
11
12 int main() {
13     // set up random-number generation
14     std::random_device rd;
15     std::default_random_engine engine{rd()};
16     std::uniform_int_distribution ints{2000, 3000};

```

Fig. 17.14 | Coordinating threads with a **std::latch** object. (Part I of 3.)

```

17 // latch that 3 threads must signal before the main thread continues
18 std::latch mainLatch{3};
19
20 // lambda representing the task to execute
21 auto task{
22     [&](std::string_view name, std::chrono::milliseconds workTime) {
23         std::cout << std::format("Proceeding with {} work for {} ms.\n",
24             name, workTime.count());
25
26         // simulate work by sleeping
27         std::this_thread::sleep_for(workTime);
28
29         // show that task arrived at mainLatch
30         std::cout << std::format("{} done; signals mainLatch.\n", name);
31         mainLatch.count_down();
32     }
33 };
34
35
36 std::vector<std::jthread> threads; // stores the threads
37 std::cout << "Main starting three jthreads.\n";
38
39 // start three jthreads
40 for (int i{1}; i < 4; ++i) {
41     // create jthread that calls task lambda,
42     // passing a task name and work time
43     threads.push_back(std::jthread{task,
44         std::format("Task {}", i),
45         std::chrono::milliseconds{ints(engine)}});
46 }
47
48 std::cout << "\nMain waiting for jthreads to reach the latch.\n\n";
49 mainLatch.wait();
50
51 std::cout << "\nAll jthreads reached the latch. Main working.\n";
52 std::cout << "Showing that mainLatch is permanently open.\n";
53 mainLatch.wait(); // latch is already open
54 std::cout << "mainLatch is already open. Main continues.\n";
55 }
```

Main starting three jthreads.

Main waiting for jthreads to reach the latch.

Proceeding with Task 3 work for 2648 ms.

Proceeding with Task 2 work for 2705 ms.

Proceeding with Task 1 work for 2024 ms.

Task 1 done; signals mainLatch.

Task 3 done; signals mainLatch.

Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.

Showing that mainLatch is permanently open.

mainLatch is already open. Main continues.

Fig. 17.14 | Coordinating threads with a `std::latch` object. (Part 2 of 3.)

```
Main starting three jthreads.
Main waiting for jthreads to reach the latch.
Proceeding with Task 2 work for 2571 ms.
Proceeding with Task 1 work for 2462 ms.
Proceeding with Task 3 work for 2248 ms.
Task 3 done; signals mainLatch.
Task 1 done; signals mainLatch.
Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.
Showing that mainLatch is permanently open.
mainLatch is already open. Main continues.
```

Fig. 17.14 | Coordinating threads with a `std::latch` object. (Part 3 of 3.)

17.13.2 C++20 `std::barrier`

Consider a simulation of the painting step in an automated automobile assembly line. Often several computer-controlled robots work together to perform a given step. Let's assume that separate threads control the cars moving along the assembly line and two robots' operations. Once the work on one car finishes, we want to reset everything, advance the assembly line and perform the work again for the next car. The preceding scenario is ideal for a `std::barrier`¹⁰⁴ (from the `<barrier>` header), which is like a **reusable latch**. Typically, a **barrier** is used for repetitive tasks in a loop:

- Each thread works, then reaches a **barrier** and waits for it to open.
- When the specified number of threads reaches the **barrier**, an optional **completion function** executes.
- The **barrier** resets its count, which unblocks the threads so they may continue executing and repeat this process.

Using a **barrier** (Fig. 17.15), let's simulate an assembly line's painting step for multiple cars. We'll also use **stop_source** and **stop_token** manually to coordinate thread cancellation when the assembly line shuts down. Again, we'll use sleeping to simulate work. Note in the output that the robots sometimes finish their "work" on each car in a different order.

```
1 // Fig. 17.15: BarrierDemo.cpp
2 // Coordinating threads with a std::barrier object.
3 #include <barrier>
4 #include <chrono>
5 #include <format>
6 #include <iostream>
7 #include <random>
8 #include <string_view>
9 #include <thread>
```

Fig. 17.15 | Coordinating threads with `std::barrier` objects. (Part 1 of 3.)

104. C++ Standard, “Thread Support Library—Coordination Types—Barriers.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/thread.barrier>.

```

10
11 int main() {
12     // simulate moving car into painting position
13     auto moveCarIntoPosition{
14         []() {
15             std::cout << "Moving next car into painting position.\n";
16             std::this_thread::sleep_for(std::chrono::seconds(1));
17             std::cout << "Car ready for painting.\n\n";
18         }
19     };
20
21     int carsToPaint{3};
22
23     // stop_source used to notify robots assembly line is shutting down
24     std::stop_source assemblyLineStopSource;
25
26     // stop_token used by paintingRobotTask to determine when to shut down
27     std::stop_token stopToken{assemblyLineStopSource.get_token()};
28
29     // assembly line waits for two painting robots to reach this barrier
30     std::barrier paintingDone{2,
31         [&]{} noexcept { // lambda called when robots finish
32             static int count{0}; // # of cars that have been painted
33             std::cout << "Painting robots completed tasks\n\n";
34
35             // check whether it's time to shut down the assembly line
36             if (++count == carsToPaint) {
37                 std::cout << "Shutting down assembly line\n\n";
38                 assemblyLineStopSource.request_stop();
39             }
40             else {
41                 moveCarIntoPosition();
42             }
43         }
44     };
45
46     // lambda that simulates painting work
47     auto paintingRobotTask{
48         [&](std::string_view name) {
49             // set up random-number generation
50             std::random_device rd;
51             std::default_random_engine engine{rd()};
52             std::uniform_int_distribution ints{2500, 5000};
53
54             // check whether the assembly line is shutting down
55             // and, if not, do the painting work
56             while (!stopToken.stop_requested()) {
57                 auto workTime{std::chrono::milliseconds(ints(engine))};
58
59                 std::cout << std::format("{} painting for {} ms\n",
60                     name, workTime.count());
61                 std::this_thread::sleep_for(workTime); // simulate work
62             }
63         }
64     };

```

Fig. 17.15 | Coordinating threads with `std::barrier` objects. (Part 2 of 3.)

```
63         // show that task woke up and arrived at continuationBarrier
64         std::cout << std::format(
65             "{} done painting. Waiting for next car.\n", name);
66
67         // decrement paintingDone barrier's counter and
68         // wait for other painting robots to arrive here
69         paintingDone.arrive_and_wait();
70     }
71
72     std::cout << std::format("{} shut down.\n", name);
73 }
74 };
75
76 moveCarIntoPosition(); // move the first car into position
77
78 // start up two painting robots
79 std::cout << "Starting robots.\n\n";
80 std::jthread leftSideRobot{paintingRobotTask, "Left side robot"};
81 std::jthread rightSideRobot{paintingRobotTask, "Right side robot"};
82 }
```

Moving next car into painting position.
Car ready for painting.

Starting robots.

Left side robot painting for 4564 ms
Right side robot painting for 2758 ms
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Left side robot painting for 4114 ms
Right side robot painting for 2860 ms
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Right side robot painting for 4730 ms
Left side robot painting for 3794 ms
Left side robot done painting. Waiting for next car.
Right side robot done painting. Waiting for next car.
Painting robots completed tasks

Shutting down assembly line

Right side robot shut down.
Left side robot shut down.

Fig. 17.15 | Coordinating threads with `std::barrier` objects. (Part 3 of 3.)

The `main` thread represents the assembly line. The program operates as follows:

- Lines 13–19 define a lambda to simulate moving the next car into the assembly line’s painting station.
- Line 21 defines `carsToPaint`—the number of cars to process in the simulation.
- We use **cooperative cancellation** (Section 17.9) to terminate the robot threads when the assembly line shuts down. `jthread`’s destructor calls its `stop_source`’s `request_stop` member function before joining the thread. The `jthread`’s task can then check its `stop_token` parameter to determine whether to terminate. In this simulation, however, we do not want the robot threads to terminate when the `jthreads` go out of scope at the end of `main`, so we handle the cancellation manually. The `stop_source` (line 24) coordinates cancellation with the robot threads. When three cars have been processed, we’ll call `request_stop` on this `stop_source` to notify the robot threads that the assembly line is shutting down.
- Line 27 gets the `stop_source`’s `stop_token`. Before painting, each robot thread will call this `stop_token`’s `stop_requested` function to check whether the assembly line is shutting down.
- Lines 30–44 define the `paintingDone barrier`, initializing it with a count of 2 (for the two painting robots) and a **completion function** (lines 31–43) that’s called when the `barrier`’s internal count reaches 0. The static local variable `count` tracks the number of cars processed so far. When the count equals `carsToPaint`, line 38 calls `request_stop` on the `stop_source` to indicate the assembly line is shutting down. Otherwise, line 41 simulates moving the next car into position.
- Lines 47–74 define the `paintingRobotTask` lambda. Lines 56–70 execute until line 56 determines that the task should stop because the assembly line is shutting down. Lines 57–65 simulate the painting work. When painting finishes, the calling `jthread` reaches the `paintingDone barrier` and calls its `arrive_and_wait` function (line 69). This decrements the `barrier`’s internal count. If the count is not 0, the calling `jthread` blocks here. If the count is 0, the `barrier`’s **completion function executes**, moving the next car into position or shutting down the assembly line. When the completion function finishes executing, the `barrier` resets its internal count and **unblocks the waiting jthreads** so they can continue executing.
- To begin the simulation, line 76 in `main` moves the first car into position, and lines 80–81 launch two `jthreads` representing the left and right painting robots.

17.14 C++20 Semaphores

Another mutual-exclusion mechanism is the **semaphore**, as described by Dijkstra in his seminal paper on cooperating sequential processes.^{105,106,107} A semaphore contains an

-
105. E. W. Dijkstra, “Cooperating Sequential Processes,” Technological University, Eindhoven, Netherlands, 1965, reprinted in F. Genuys, ed., *Programming Languages*, pp. 43–112. New York: Academic Press, 1968.
 106. Harvey Deitel, Paul Deitel and David Choffnes, Chapter 5, “Asynchronous Concurrent Execution.” *Operating Systems*, 3/e, pp. 227–233. Upper Saddle River, NJ: Prentice Hall, 2004.
 107. “Semaphore (Programming).” Accessed April 18, 2023. [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming)).

integer value representing the maximum number of concurrent threads that can access a shared resource, such as **shared mutable data**. Once initialized, that integer can be accessed and altered by only two operations, *P* and *V*. These are short for the Dutch words *proberen*, meaning “to test,” and *verhogen*, meaning “to increase.”¹⁰⁸ A thread calls the *P* operation (also called the **wait operation**) when it wants to **enter a critical section** and calls the *V* operation (also called the **signal operation**) when it wants to **exit a critical section**. Once the maximum number of threads are operating in the **critical section**, other threads trying to enter must wait. *P* and *V* are abstractions that encapsulate the details of mutual exclusion implementations. They can support any number of cooperating threads. C++’s semaphore classes call these operations **acquire** and **release**. For a nice discussion of semaphore fundamentals, check out Allen B. Downey’s *The Little Book of Semaphores*.¹⁰⁹

C++20’s `<semaphore>` header contains features for implementing **counting semaphores** and **binary semaphores**:

- A `std::counting_semaphore` implements the semaphore concept, which is lower level than `std::latch` and `std::barrier`, but still higher level than mutexes, locks, condition_variables and atomics.^{110,111} A `counting_semaphore` can allow multiple threads to access a shared resource. Its constructor initializes its internal integer counter. When a thread **acquires the semaphore**, the internal counter decrements by one. If the counter reaches zero, a thread attempting to **acquire the semaphore** will block until the count increases to indicate that the shared resource is available. When a thread **releases the semaphore**, the internal counter increments by one (by default) and threads waiting to **acquire the semaphore** unblock.
- A `std::binary_semaphore` is simply a `counting_semaphore` with a count of 1 and can be used like a `std::mutex`.¹¹²

Producer–Consumer Using C++20 `std::binary_semaphore`

The C++ standard says semaphores “are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.”¹¹³ Figure 17.16 reimplements class `SynchronizedBuffer` from Section 17.6 using **binary_semaphores** for mutual exclusion. This gives you a neat, simple, higher-level way to replace lower-level mutex code. We reuse Fig. 17.8’s `main` function, so we do not repeat it here.



-
108. “Semaphore (Programming)—Operation Names.” Accessed April 18, 2023. [https://en.wikipedia.org/wiki/Semaphore_\(programming\)#Operation_names](https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names).
109. Allen B. Downey, *The Little Book of Semaphores* (Version 2.2.1), Section “3.6.4 Barrier Solution,” 2016. Green Tea Press. <https://greenteapress.com/seahorses/LittleBookOfSemaphores.pdf>. License: <http://creativecommons.org/licenses/by-nc-sa/4.0>. Thanks to one of our reviewers, Anthony Williams, for pointing us to Downey’s work.
110. C++ Standard, “Thread Support Library—Semaphore.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/thread.sema>.
111. “`std::counting_semaphore`, `std::binary_semaphore`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/counting_semaphore.
112. “`std::counting_semaphore`, `std::binary_semaphore`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/thread/counting_semaphore.
113. C++ Standard, “32.7 Semaphore.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/thread.sema>.

```

1 // Fig. 17.16: SynchronizedBuffer.h
2 // SynchronizedBuffer using two binary_semaphores to
3 // maintain synchronized access to a shared mutable int.
4 #pragma once
5 #include <format>
6 #include <iostream>
7 #include <semaphore>
8 #include <string>
9
10 using namespace std::string_literals;
11
12 class SynchronizedBuffer {
13 public:
14     // place value into m_buffer
15     void put(int value) {
16         // acquire m_produce semaphore to be able to write to m_buffer
17         m_produce.acquire(); // blocks if it's not the producer's turn
18
19         m_buffer = value; // write to m_buffer
20         m_occupied = true;
21
22         std::cout << std::format("{:<40}{}\t{}\n",
23             "Producer writes "s + std::to_string(value),
24             m_buffer, m_occupied);
25
26         m_consume.release(); // allow consumer to read
27     }
28
29     // return value from m_buffer
30     int get() {
31         int value; // will store the value returned by get
32
33         // acquire m_consume semaphore to be able to read from m_buffer
34         m_consume.acquire(); // blocks if it's not the consumer's turn
35
36         value = m_buffer; // read from m_buffer
37         m_occupied = false;
38
39         std::cout << std::format("{:<40}{}\t{}\n",
40             "Consumer reads "s + std::to_string(m_buffer),
41             m_buffer, m_occupied);
42
43         m_produce.release(); // allow producer to write
44         return value;
45     }
46 private:
47     std::binary_semaphore m_produce{1}; // producer can produce
48     std::binary_semaphore m_consume{0}; // consumer can't consume
49     bool m_occupied{false};
50     int m_buffer{-1}; // shared by producer and consumer threads
51 };

```

Fig. 17.16 | SynchronizedBuffer using two binary_semaphores to maintain synchronized access to a shared mutable int. (Part I of 2.)

Operation	Buffer	Occupied
Producer writes 1	1	true
Consumer reads 1	1	false
Producer writes 2	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Consumer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Producer writes 9	9	true
Consumer reads 9	9	false
Producer writes 10	10	true
Producer done producing		
Terminating Producer		
Consumer reads 10	10	false
Consumer read values totaling 55		
Terminating Consumer		

Fig. 17.16 | SynchronizedBuffer using two `std::binary_semaphores` to maintain synchronized access to a shared mutable `int`. (Part 2 of 2.)

Figure 17.16 uses two `std::binary_semaphores` to coordinate the producer and consumer threads:

- `m_produce` (line 47) is initialized with the count 1, indicating that the producer initially can produce a value because `m_buffer` is empty, and
- `m_consume` (line 48) is initialized with the count 0, indicating that the consumer initially cannot consume a value because `m_buffer` is empty.

This example uses `m_occupied` (line 49) for output purposes only—it does not play a role in this example’s thread synchronization.

SynchronizedBuffer Member Function `put`

`SynchronizedBuffer`’s logic is simplified with `std::binary_semaphores`. When the producer thread calls `put` (defined at lines 15–27), line 17 calls `m_produce`’s **acquire member function**. If `m_produce`’s count is 0, the producer thread will be blocked until the count is 1. If the count is 1, the producer thread acquires the semaphore, decreasing its count to 0, and writes a new value into `m_buffer`. The producer cannot produce again until `m_produce`’s count is 1, which will occur only when the consumer consumes the buffer’s value.

When the producer is done updating the buffer, line 26 calls `m_consume`’s **release member function** to increment that semaphore’s count to 1. If a consumer thread is blocked waiting to acquire `m_consume`, it unblocks so it can proceed. Otherwise, when a consumer tries to call `get`, it can immediately acquire `m_consume` and proceed.

SynchronizedBuffer Member Function `get`

When the consumer thread calls `get` (defined at lines 30–45), line 34 calls `m_consume`'s **acquire member function**. If `m_consume`'s count is 0, the consumer thread blocks until the count is 1. If the count is 1, the consumer thread **acquires** `m_consume`, decreasing its count to 0, then reads `m_buffer`'s current value. The consumer cannot consume again until `m_consume`'s count is 1, which will occur only when the producer writes the next value into the buffer.

When the consumer is done reading from the buffer, line 43 calls `m_produce`'s **release member function** to increment that semaphore's count to 1. If a producer thread is blocked waiting to acquire `m_produce`, it unblocks so it can proceed. Otherwise, when a producer tries to call `put`, it can immediately **acquire** `m_produce` and proceed.

17.15 C++23: A Look to the Future of C++ Concurrency

Many new concurrency features are being considered for future C++ versions. Here, we overview several of them and provide links to where you can learn more.^{114,115}

17.15.1 Parallel Ranges Algorithms

The C++ Standard Committee is working on parallelized versions of C++20's `std::ranges` algorithms. The proposal, "A Plan for C++23 Ranges," indicates that there are implementation issues tied to executors.¹¹⁶

17.15.2 Concurrent Containers

You've seen various **non-concurrent containers** throughout this book, and in this chapter, you studied a `CircularBuffer` class in which we implemented concurrent access via `std::mutex`, `std::unique_lock` and `std::condition_variable`. Generally, rather than creating your own concurrent containers, as we did, it's good practice to use preexisting ones that manage synchronization for you—such as concurrent queues and concurrent maps (also called concurrent hash tables or concurrent dictionaries). These are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls. Such containers might be included in the C++23 standard library. For now, you'll need to use third-party libraries, such as the containers in the **Google Concurrency Library (GCL)**¹¹⁷ or the **Microsoft Parallel Patterns Library**.¹¹⁸ For more information on concurrent queues, see the "C++ Concurrent Queues" proposal.¹¹⁹ The proposal indicates that the **concurrent queue reference implementations** are pro-

114. "C++23." Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B23>.

115. Ville Voutilainen, "To Boldly Suggest an Overall Plan for C++23," November 25, 2019. Accessed April 18, 2023. <https://wg21.link/p0592>.

116. Barry Revzin, Conor Hoekstra and Tim Song, "A Plan for C++23 Ranges," October 14, 2020. Accessed April 18, 2023. <https://wg21.link/p2214>.

117. Alasdair Mackintosh, "Google Concurrency Library (GCL)." Accessed April 18, 2023. <https://github.com/alasdairmackintosh/google-concurrency-library>.

118. "Parallel Patterns Library (PPL)—Parallel Containers and Objects." Accessed April 18, 2023. <https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-containers-and-objects>.

119. Lawrence Crowl and Chris Mysen, "C++ Concurrent Queues." Accessed April 18, 2023. <https://wg21.link/p0260>.

vided by the GCL's **buffer_queue** and **lock_free_buffer_queue** classes. For more information on concurrent maps, see the C++ Standards Committee proposals:

- “Concurrent Associative Data Structure with Unsynchronized View”¹²⁰ and
- “Concurrent Map Customization Options (SG1 Version)”¹²¹

and the **concurrent map** reference implementation at

<https://github.com/BlazingPhoenix/concurrent-hash-map>

17.15.3 Other Concurrency-Related Proposals

The following additional concurrency-related C++ Standards Committee proposals are being considered for C++23 and beyond:

- The “Hazard Pointers”¹²² and “Concurrent Data Structures: Read-Copy-Update”¹²³ proposals introduce features for safely reclaiming resources shared among threads.
- The “`apply()` for `synchronized_value<T>`”¹²⁴ proposal enhances an earlier proposal that introduced `synchronized_value<T>`, which automatically uses a `mutex` to synchronize concurrent access to an object of type `T`. The proposed `apply` function would receive as arguments a function and one or more `synchronized_value<T>` objects. It would then call its function argument on each `synchronized_value<T>`, using the associated `mutex` to synchronize access from concurrent threads.

17.16 Wrap-Up

This chapter presented modern standardized C++ concurrency features for enhancing application performance on multi-core systems. We compared sequential, concurrent and parallel execution, and presented a sample thread-life-cycle diagram. We discussed why concurrent programming is complex and error-prone. Generally, developers should prefer the simpler, more convenient, high-level, prebuilt concurrency capabilities that we emphasized in this chapter.

We showed that high-level parallel algorithms automatically parallelize tasks for better performance. We used the `<chrono>` header’s timing capabilities to profile sequential and parallel algorithm performance on multi-core systems and showed the parallel `std::sort`

120. Sergey Murylev, Anton Malakhov and Antony Polukhin, “Concurrent Associative Data Structure with Unsynchronized View,” June 13, 2019. Accessed April 18, 2023. <http://wg21.link/p0652>.

121. David Goldblatt. “Concurrent Map Customization Options (Sg1 Version),” June 16, 2019. Accessed April 18, 2023. <https://wg21.link/P1761>.

122. Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O’Dwyer, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn and Jens Maurer, “Hazard Pointers,” April 9, 2021. Accessed April 18, 2023. <https://wg21.link/p1121>.

123. Paul McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O’Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kaminski and Jens Maurer, “Concurrent Data Structures: Read-Copy-Update,” May 14, 2021. Accessed April 18, 2023. <https://wg21.link/p1122>.

124. Anthony Williams, “`apply()` for `synchronized_value<T>`,” March 2, 2017. Accessed April 18, 2023. <https://wg21.link/p0290>.

algorithm's significant performance improvement for big enough data sets (when called with the `std::execution::par` execution policy) over its sequential counterpart. We also compared the `std::transform` algorithm's performance using the `std::execution::par` (parallel) and `std::execution::unseq` (vectorized) execution policies.

We executed tasks in separate threads via C++20's `std::jthread` class, which the C++ Core Guidelines recommend using in preference to `std::thread`. We used `jthread`'s integrated cooperative-cancellation support to enable threads to terminate gracefully so that they can complete critical work and correctly release resources.

We introduced producer-consumer relationships, which are popular in concurrent programming, and demonstrated the problems with a producer thread and a consumer thread simultaneously accessing shared mutable data without synchronization. We corrected those problems by synchronizing access to shared mutable data using low-level concurrency primitives `std::mutex`, `std::condition_variable` and `std::unique_lock`. Next, we used these primitives to implement a synchronized circular buffer to minimize producer-consumer waiting and increase performance.

We used `std::async` and `std::future` to implicitly create threads, execute tasks asynchronously and communicate their results back to the thread that called `async`. We discussed how `async` uses a `std::promise` "under the hood" for inter-thread communication to return a task's result, or an exception, to the thread that called `async`.

We introduced atomic types, which enable threads to **share mutable data conveniently without explicit synchronization and locking**. We overviewed several C++20 atomics enhancements and mentioned that the C++ standard library implementations for Visual C++, g++ and clang++ use atomics to implement the higher-level C++20 primitives `std::latch`, `std::barrier` and semaphores.

We demonstrated the `std::latch` and `std::barrier` thread-coordination primitives, showing that they do not require mutexes and locks to work correctly. Next, we used C++20 semaphores without mutexes, locks and condition variables to synchronize access to shared mutable data.

Finally, we mentioned several high-level concurrency capabilities being considered for C++23 and later versions, including concurrent containers and parallel versions of the range-based algorithms. **Most programmers should prefer prebuilt concurrent containers, such as concurrent queues and maps, that encapsulate synchronization.** These help avoid the common traps and pitfalls of using low-level synchronization primitives. Such concurrent containers are not yet part of the C++ standard library, so you'll need to use existing third-party libraries, such as the Google Concurrency Library and the Microsoft Parallel Patterns Library.

In the next chapter, we'll present a detailed treatment of the concurrency feature coroutines—the last of C++20's "big four" features (which also include ranges, concepts and modules). You'll use a high-level, library-based approach to conveniently create coroutines, enabling sophisticated concurrent programming with a simple sequential-like coding style.

Self-Review Exercises

17.1 Fill in the blanks in each of the following statements:

- Two tasks that are operating _____ are both making progress, possibly in small increments and not necessarily simultaneously.

- b) The C++ standard says, “_____ is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.”
- c) One approach to thread safety is using only _____ data.
- d) You can use the timing features from the _____ header to demonstrate the performance improvement of one algorithm over another.
- e) Each parallel algorithm overload requires as its first parameter a(n) _____ indicating whether to parallelize a task and, if so, how to do it.
- f) The _____ execution policy indicates that the algorithm should try to execute portions of its work simultaneously on multiple cores.
- g) The parallel algorithm _____ produces a single value from a range of values.
- h) Operating systems can employ a technique called _____ to prevent starvation—as a thread waits in the ready state, the operating system gradually increases the thread’s priority to ensure that it will eventually run.
- i) A thread is _____ if it is waiting for a particular event that will not occur.
- j) The four necessary conditions that Coffman, Elphick and Shoshani proved are necessary for deadlock to exist are mutual exclusion, wait for, no preemption and _____.
- k) Deadlock and indefinite postponement each involve some form of _____.
- l) All operations on shared mutable data accessed by concurrent threads must be guarded with a(n) _____ to prevent corruption.
- m) When a multithreaded application needs to terminate, it’s good practice to shut down threads that are still performing tasks, so they can release resources they’re using. C++20 added _____ to enable programs to notify threads when it’s time for them to terminate. A task executing in a thread can watch for such notifications, then complete critical work, release resources and terminate.
- n) The _____ header contains features that enable you to execute asynchronous tasks and receive the results of those tasks when they finish executing.
- o) If multiple threads need access to an asynchronous task’s result, you can use _____ objects.
- p) Typically, a `barrier` is used for repetitive tasks in a loop: Each thread works, then reaches a barrier and waits for it to open. When the specified number of threads reaches the `barrier`, an optional _____ executes. The `barrier` resets its count, which unblocks the threads so they may continue executing and repeat this process.
- q) In Dijkstra’s seminal paper on cooperating sequential processes, the `P` operation is also called the _____ operation, and the `V` operation is also called the _____ operation.
- r) A `std::binary_semaphore` is simply a `counting_semaphore` with a count of _____ and can be used like a `std::mutex`.
- 17.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- a) Concurrency is a subset of parallelism.
- b) Programs can have multiple threads of execution, each with its own function-call stack and program counter, allowing it to execute concurrently with other threads.
- c) Parallel algorithms always improve performance over sequential algorithms.

- d) We cannot predict the order in which tasks will start executing, even if we know the order in which they were created and started. This is one of the challenges of multithreaded programming.
- e) Print spooling is a common example of a producer–consumer relationship. A printer is an exclusive resource. Although a printer might not be available when you want to print from an application (the producer), you can still “complete” the print task. The data is temporarily stored (called spooling) until the printer becomes available.
- f) A thread should never sleep while holding a lock in a real application.
- g) A circular buffer never becomes full.
- h) Before using the `++` operator on an atomic object, you must synchronize access to the object.

17.3 (*Mutual Exclusion*) Discuss how mutually exclusive access to shared mutable data may be performed with critical sections and `mutexes`.

Answers to Self-Review Exercises

17.1 a) concurrently. b) a thread of execution. c) immutable. d) `<chrono>`. e) execution policy. f) `std::execution::par`. g) reduce. h) aging. i) deadlocked. j) circular wait. k) waiting. l) lock. m) cooperative cancellation. n) `<future>`. o) `std::shared_future`. p) completion function. q) wait, signal. r) 1.

17.2 a) False. Actually, parallelism is a subset of concurrency. b) True. c) False. Actually, you cannot simply assume that using parallel algorithms will improve performance. Sometimes parallel algorithms actually perform worse than the corresponding sequential algorithms. This is especially true when processing small numbers of elements and when using non-random-access iterators. In these cases, the overhead of parallelization may outweigh the benefits of parallel performance. d) True. e) True. f) True. g) False. Actually, even with a circular buffer, a producer thread could fill the buffer, forcing the producer to wait until a consumer consumed a value to free an element in the buffer. h) False. Actually, operator `++` is one of a limited number of operations you can perform on atomic objects. When one thread is executing `++` on an atomic object, the increment is guaranteed to complete before another thread can modify or view that atomic object’s value.

17.3 A common way to implement mutually exclusive access to shared mutable resources is by creating critical sections. These synchronized blocks of code execute atomically using features from the `<mutex>` header. A `std::mutex` can be owned by only one thread at a time. A thread that requires exclusive access to a resource must first acquire a lock on a `mutex`—typically at the beginning of a block of code. Other threads attempting to perform operations that require the same `mutex` will be blocked until the first thread releases the lock—typically at the end of a block of code. At that point, the blocked threads may attempt to acquire the lock and proceed with the operation.

Exercises

17.4 Fill in the blanks in each of the following statements:

- a) Two tasks that are operating _____ are truly executing simultaneously.

- b) When threads share mutable (modifiable) data, you must ensure that they do not corrupt it, which is known as making the code _____.
- c) Computer processing power continues to increase, but Moore's law has essentially expired, so hardware vendors now rely on _____ processors for better performance.
- d) An object of type _____ is recommended for timing operations.
- e) The _____ execution policy indicates that an algorithm can be parallelized and vectorized.
- f) The parallel algorithm execution policies are just suggestions—compilers can ignore them or handle them differently. For example, if a compiler targets hardware that does not support vectorization, the compiler might default the `par_unseq` policy to _____.
- g) To take full advantage of multi-core architecture, you need to write _____ applications so separate threads can run in parallel on a multi-core system when sufficient cores are available.
- h) A **ready** thread enters the _____ state (i.e., begins executing) when the operating system assigns it to a processor—also known as dispatching the thread.
- i) The necessary condition for a deadlock to exist described by: “Once a thread has obtained a resource, the system cannot remove it from the thread’s control until the thread has finished using the resource” is _____.
- j) In _____, a thread that is not deadlocked could wait for an event that might never occur or might occur unpredictably far in the future because of biases in the system’s resource-scheduling policies.
- k) The C++ standard provides two classes for launching concurrent tasks in an application—`std::thread` and `std::jthread`. You should prefer _____.
- l) After scheduling tasks to execute, you typically want to wait for them to complete—for example, to use their results. You wait for a `jthread` to complete its task by “joining the thread” with a call to its _____ function.
- m) In a multithreaded producer-consumer relationship, a producer thread generates data, placing it in a shared object called a buffer. A consumer thread reads data from the buffer. This relationship requires _____ to ensure that values are produced and consumed correctly.
- n) Condition variables can be used to make a thread _____ while a condition is not satisfied then to notify a waiting thread to _____ when a condition is satisfied.
- o) Using a(n) _____ buffer, you can minimize waiting among concurrent threads that share resources and operate at the same average speeds. Such a buffer provides a fixed number of cells into which the producer can write values and from which the consumer can read those values. Internally, the buffer manages the producer’s writes into the buffer and the consumer’s reads from the buffer elements in order, beginning at the first cell and moving toward the last. When the buffer reaches its last element, it “wraps around” to the first element and continues from there.

- p) Function `std::async` returns an object of class template _____, which enables inter-thread communication between the thread that calls `async` and the task `async` executes. The C++ Core Guidelines recommend using a **future** to return a result from an asynchronous task.
- q) A `std::latch` is a single-use gateway in your code that remains closed until a specified number of threads reach the latch. At that point, the gateway _____. The gateway serves as a one-time synchronization point, allowing threads to wait until a specified number of threads reach that point.
- r) Consider a simulation of the painting step in an automated automobile assembly line. Often several computer-controlled robots work together to perform a given step. Let's assume that cars moving along the assembly line and two robots' operations are all individually controlled by threads. Once the work on one car finishes, we want to reset everything, advance the assembly line and perform the work again for the next car. The preceding scenario is ideal for a(n) _____, which is like a reusable latch.
- 17.5** State whether each of the following is *true* or *false*. If *false*, explain why.
- For multithreading to be effective, you must have a multiple-processor system.
 - For C++11 and C++14, the C++ Standards Committee defined mostly high-level primitives. These capabilities were then used to build lower-level C++17 and C++20 features. They're also being used to implement the lower-level features coming in C++23 and later.
 - Unlike the `accumulate` algorithm, the `reduce` algorithm does not guarantee the order in which elements are processed, which enables `reduce` to be parallelized for better performance.
 - When concurrent threads share mutable data and that data is modified by one or more of them, indeterminate results may occur. In such cases, the program's behavior cannot be trusted.
 - Immutable data can be safely accessed by only one thread at a time.
 - The C++ Core Guidelines say to minimize the duration of critical sections—that is, the amount of time an object is “locked.” Avoid performing I/O, lengthy calculations and operations that do not require synchronization while holding a lock.
 - With a circular buffer: If a producer temporarily operates faster than a consumer, the producer can write additional values into the extra buffer cells, if any are available. This enables the producer to keep busy even though the consumer is not ready to retrieve the current value being produced. If the consumer temporarily operates faster than the producer, the consumer can read additional values (if there are any) from the buffer. This enables the consumer to keep busy even though the producer is not ready to produce additional values.
 - Generally, you should create your own concurrent containers.

17.6 (*Multithreading Terms*) Define each of the following terms.

- thread
- multithreading
- runnable* state
- timed waiting* state
- preemptive scheduling

- f) producer/consumer relationship
- g) quantum

Discussion Exercises

17.7 (Blocked State) List the reasons for entering the **blocked** state. For each of these, describe how the program will normally leave the **blocked** state and enter the **runnable** state.

17.8 (Deadlock and Indefinite Postponement) Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

17.9 (Bounded Buffer: A Real-World Example) Describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer. In particular, discuss how the designers might choose the size of the off-ramp.

17.10 (Optimal Size of a Circular Buffer) Discuss how to determine the optimal size for a circular buffer.

17.11 (Readers and Writers) In the Readers and Writers protocol, we must ensure that they perform their tasks only when it's their turn. Consider the following: If a reader is reading when a writer arrives, the writer must wait for the lock to become available. Also, any additional readers that subsequently arrive must wait until the currently waiting writer executes and notifies them that it has finished writing. What would happen to the waiting writer if a stream of arriving readers were allowed to read?

17.12 (Discussion: Producer/Consumer Relationship) Concisely trace the evolution of the producer/consumer relationship and the related readers and writers relationship through Sections 17.5–17.8. For each section, y briefly state:

- a) The problem that implementation is trying to solve.
- b) The approach to solving the problem.
- c) Any notable weaknesses and/or subtleties in solution correctness, performance or complexity that become apparent.

Code Exercises

17.13 (Profiling Sequential and Parallel Algorithms) In Fig. 17.1, we sorted 100 million integers using the `std::sort` algorithm's sequential and parallel versions. We displayed the time each operation took to complete to demonstrate the performance difference between them. Modify Fig. 17.1 to fill a one billion element `vector` with random `int` values in the range 1 to 1000. Use the `std::reduce` algorithm's sequential and parallel versions to sum the elements. Display the amount of time each operation takes to complete. Use `OLL` (the `long long` value 0) to initialize `std::reduce` algorithm's sum, and use the `std::plus{}` function object to enable `std::reduce` to calculate the sum.

17.14 (Timing 60,000,000 Die Rolls) Figure 6.7 used random-number generation to roll a six-sided die 60,000,000 times. Modify that program to calculate the amount of time your system requires to summarize the 60,000,000 die rolls.

17.15 (Parallelizing 60,000,000 Die Rolls) Modify Exercise 17.14 to parallelize the 60,000,000 die rolls as follows:

- Define the function `dieRollingTask` that receives as arguments the number of dice to roll and a reference to a seven-element `std::array`. The function should roll the specified number of dice and summarize their frequencies in the array. As in Fig. 6.7, we'll ignore element 0 and use elements 1–6 as the frequency counters for the faces 1–6.
- Determine how many cores your computer's processor has by calling `jthread`'s static function `std::jthread::hardware_concurrency()`.
- Create a `vector` with the same number of elements as the number of cores on your computer. Each element of this `vector` should be a seven-element `array`, which we'll use as frequency counters for the die faces (again, ignoring element 0 of each).
- Using the techniques you learned in Section 17.4, launch the same number of `std::jthreads` as your system has cores. Each `std::jthread` should execute the `dieRollingTask` for a number of die rolls determined by calculating 60,000,000 divided by your system's number of cores—so, if your system has four cores, each `std::jthread` will perform 15,000,000 die rolls.

When the `std::jthreads` complete, calculate each face's total frequency by summing each array's corresponding elements. Calculate the total time your system requires to summarize the parallelized 60,000,000 die rolls. How did the time compare to Exercise 17.14?

17.16 (Launching Tasks with `std::async`) Section 17.4's program created several `std::jthread` objects to execute `printTask` calls in separate threads. Modify the program so that, rather than manually creating `std::jthreads`, it uses `std::async` to execute the `printTask` calls in separate threads.

17.17 (Atomic Counters) Modify your solution to Exercise 17.15 so that, rather than maintaining a separate array of counters for each thread, the program maintains an array of `std::atomic<int>` counters that all the threads share. How does the performance of this solution compare to that of Exercise 17.15?

17.18 (Circular Buffer with Semaphores) Use a `counting_semaphore` (Section 17.14) to reimplement the `CircularBuffer` class (Section 17.7). Test your updated `CircularBuffer` class with the program of Fig. 17.10.

17.19 (Latches) Consider a parallel lowercase-letter-count algorithm that

- launches several worker threads to count the occurrences of each lowercase letter (a through z) in portions of a large array (ignore all other characters),
- waits for the worker threads to complete, then
- combines the results from the worker threads into a single a-through-z distribution of counts.

Assume the algorithm uses three worker threads, each processing about a third of the array. The algorithm should use a `std::latch` object to wait until the workers are done. Base your solution to this exercise on Fig. 17.14's code.

17.20 (Barriers) Consider a simulation of the door-assembly step in an automated four-door-truck assembly line. Assume that separate threads control the cars moving along the assembly line and four robots operations. Each robot attaches one of the four doors to the

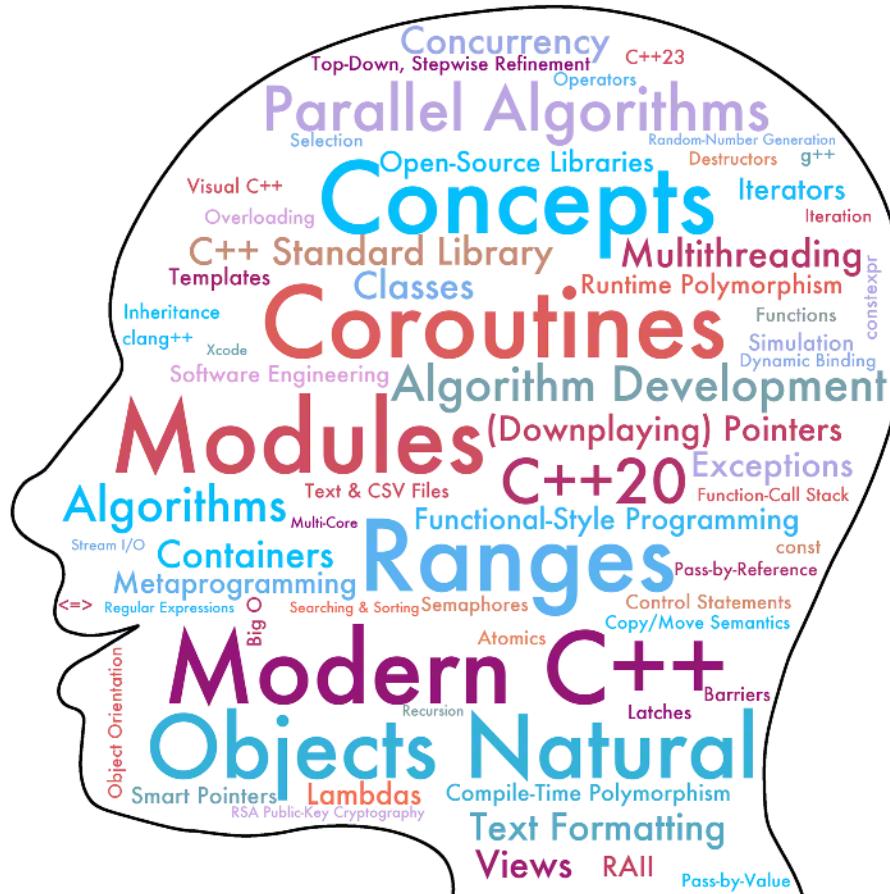
truck. Once all four doors have been attached to the truck, reset everything, advance the assembly line and perform the work again on the next truck.

Each thread works, then reaches a barrier and waits for it to open. When the specified number of threads reaches the barrier, an optional completion function executes. The barrier resets its count, which unblocks the threads so they may continue executing, then repeat this process.

As in Fig. 17.15, use `stop_source` and `stop_token` manually to coordinate thread cancellation when the assembly line shuts down. Use sleeping to simulate work. Note in the output that the robots sometimes finish their work on each car in different order. Base your solution to this exercise on Fig. 17.15's code.

This page intentionally left blank

C++20 Coroutines



Objectives

In this chapter, you'll:

- Understand what coroutines are and how they're used.
- Use keyword `co_yield` to suspend a generator coroutine and return a result.
- Use the `co_await` operator to suspend a coroutine while it waits for a result to become available.
- Use a `co_return` statement to terminate a coroutine and return its result, or simply control, to its caller.
- Use the open-source `generator` library to simplify creating a generator coroutine with `co_yield`.
- Use the open-source `concurrentpp` library to simplify creating coroutines with `co_await` and `co_return`.
- Become aware of coroutine capabilities being contemplated for future C++ versions.

Outline

- | | |
|--|---|
| 18.1 Introduction
18.2 Coroutine Support Libraries
18.3 Installing the <code>concurrency</code> and <code>generator</code> Libraries
18.4 Creating a Generator Coroutine with <code>co_yield</code> and the <code>generator</code> Library
18.5 Launching Tasks with <code>concurrency</code> | 18.6 Creating a Coroutine with <code>co_await</code> and <code>co_return</code>
18.7 Low-Level Coroutines Concepts
18.8 Future Coroutines Enhancements
18.9 Wrap-Up
Exercises |
|--|---|

18.1 Introduction

When a program contains a long-running task, it's common for a function that you call **synchronously**—that is, performing tasks one after another—to launch the long-running task **asynchronously**. Typically, the program would provide that asynchronous task with a **callback function** (a function, a lambda or a function object) to call when the task completes. This coding style is simplified with C++20 **coroutines**—the last of C++20's “big four” features (ranges, concepts, modules and coroutines) we cover in this book.



A **coroutine**¹ is a function that can **suspend execution and be resumed later**. The mechanisms that

- suspend a coroutine and return control to its caller and
- continue a suspended coroutine's execution later

are handled entirely by code that's written for you by the compiler. You'll see that a function containing any of the keywords `co_await`, `co_yield` or `co_return` is a coroutine.

Coroutines enable you to do concurrent programming with a simple sequential-like coding style. This capability requires sophisticated infrastructure. You can write the infrastructure yourself, but doing so is complex, tedious and error-prone. Instead, most experts agree **programmers should use high-level coroutine support libraries**, which is the approach we demonstrate. The open-source community has created several experimental libraries for developing coroutines quickly and conveniently. Section 18.2 lists several and provides the rationale for the two we use in our coroutines examples.

Coroutine Use Cases

Some coroutines use cases² include:

- web servers handling requests, enabling a function's execution to span multiple animation frames in game programming, consuming data once it becomes available (such as the results of long-running calculations; Section 18.5), data coming from devices in the Internet of Things (IoT) or downloads of large files;³

1. The term “coroutine” was coined by Melvin Conway in 1958. “Coroutine.” Accessed April 18, 2023. <https://en.wikipedia.org/wiki/Coroutine>.
2. Geoffrey Romer, Gor Nishanov, Lewis Baker and Mihail Mihailov, “Coroutines: Use-Cases and Trade-Offs,” February 19, 2019. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1493r0.pdf>.
3. “What Are Use Cases for Coroutines?” Accessed April 18, 2023. <https://stackoverflow.com/questions/303760/what-are-use-cases-for-coroutines>.

- lazily computed sequences (known as **generators**; Section 18.4) that produce one value at a time on demand;⁴
- event-driven coding without callback functions⁵—for example, simulations, user interfaces, servers, games, non-blocking I/O;⁶
- cooperative multitasking;^{7,8}
- structured concurrency;⁹
- reactive streams programming;¹⁰ and
- implementing state machines.¹¹

18.2 Coroutine Support Libraries

Coroutines require various supporting classes with numerous member functions and nested types. Creating these yourself is cumbersome, complex and error-prone. The C++20 standard library includes only the low-level primitives that library writers need to build coroutines support libraries. Lewis Baker, who has worked on several such libraries, said that these primitives “can be thought of as a low-level assembly language for coroutines.”¹² Most developers will prefer to work with a coroutine support library. Standard library coroutine support is expected in C++23.^{13,14}

In the interim, the open-source community has created several non-standard experimental coroutine support libraries. Here are the ones we considered for this presentation:

-
4. “Coroutines,” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/coroutines>.
 5. David Mazières, “My Tutorial and Take on C++20 Coroutines,” February 2021. Accessed April 18, 2023. <https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html>.
 6. Techmunching, “Coroutines and Their Introduction in C++,” May 30, 2020. Accessed April 18, 2023. <https://techmunching.com/coroutines-and-their-introduction-in-c/>.
 7. Rainer Grimm, “C++20: More Details to Coroutines,” March 27, 2020. Accessed April 18, 2023. <http://modernescpp.com/index.php/component/content/article/54-blog/c-20/488-c-20-coroutines-more-details>.
 8. Bobby Priambodo, Cooperative vs. Preemptive: a Quest to Maximize Concurrency Power,” September 3, 2019. Accessed April 18, 2023. <https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe>.
 9. Lewis Baker, “Structured Concurrency: Writing Safer Concurrent Code with Coroutines and Algorithms,” October 14, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=1Wy5sq3s2rg>.
 10. Jeff Thomas, “Exploring Coroutines,” April 7, 2021. Accessed April 18, 2023. <https://blog.coffeetocode.com/2021/04/exploring-coroutines/>.
 11. Steve Downey, “Converting a State Machine to a C++ 20 Coroutine,” June 29, 2021. Accessed April 18, 2023. <https://www.youtube.com/watch?v=Z8jHi9Cs6Ug>.
 12. Lewis Baker, “C++ Coroutines: Understanding Operator `co_await`,” November 17, 2017. Accessed April 18, 2023. <https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await>.
 13. Ville Voutilainen, “To Boldly Suggest an Overall Plan for C++23,” November 25, 2019. Accessed April 18, 2023. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html>.
 14. “C++23.” Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B23>.

- Lewis Baker's `cppcoro`¹⁵ is frequently mentioned in many books, videos and blog posts but is no longer supported.¹⁶
- Facebook's `folly::coro` is a subset of its popular and large `folly` open-source C++ utilities library.¹⁷ Lewis Baker is part of the Facebook team responsible for `folly::coro`. This library's documentation indicates that `folly::coro` is experimental. Unfortunately, `folly::coro` cannot be installed independently of `folly`.
- David Haim's `concurrentpp`¹⁸ is actively maintained. This library enables you to conveniently develop coroutines using `co_await`, `co_return` and `co_yield`.
- Sy Brand's `generator`¹⁹ is a header-only library for developing coroutines that return one value at a time via `co_yield`. A generator coroutine produces values **on demand**, known as **lazy evaluation**. This is in contrast to **greedy evaluation**—for example, the `std::ranges::generate` algorithm (Section 14.4.1) **immediately** generates values and places them into a range, such as a `vector`. For large numbers of items, creating a range can take substantial memory and time. If a program does not need all the values at once, generators can reduce your program's memory consumption and improve performance. Generators also can define infinite sequences, such as the Fibonacci sequence (Section 18.4) or prime numbers.²⁰

Perf 

For our coroutines presentation, we use `concurrentpp` and `generator`.²¹ `concurrentpp` installs easily, is actively maintained, and has clear documentation with many code examples. It provides **high-level concurrency features**, including

- **tasks** for executing functions **asynchronously**,
- **executors** for **scheduling tasks**, possibly executing them in separate threads,
- **timers** for **performing tasks in the future** and
- various **utility functions**.

In Section 18.4's example, we use the `generator` library to implement a Fibonacci generator coroutine that generates the next Fibonacci value in sequence on demand and returns it to the caller via `co_yield`. Section 18.5's example introduces `concurrentpp` **tasks** and **executors**—standardized versions of each are expected in C++23. Section 18.6's example uses these to implement a coroutine demonstrating `co_await` and `co_return`.

15. Lewis Baker, “`cppcoro`.” Accessed April 18, 2023. <https://github.com/lewissbaker/cppcoro>.

16. In a September 18, 2021 email interaction, Mr. Baker indicated that `cppcoro` is no longer supported. He now works on Facebook's experimental `folly::coro` and `libunifex` libraries.

17. “facebook/folly.” Accessed April 18, 2023. <https://github.com/facebook/folly>.

18. David Haim, “`concurrentpp`.” Accessed April 18, 2023. <https://github.com/David-Haim/concurrentpp>.

19. Sy Brand (C++ Developer Advocate, Microsoft), “`generator`.” Accessed April 18, 2023. <https://github.com/TartanLlama/generator>.

20. Visual C++ provides a `std::experimental::generator` implementation in its `<experimental/generator>` header.

21. Thank you to Anthony Williams (<https://www.linkedin.com/in/anthonyajwilliams>)—author of *C++ Concurrency in Action, Second Edition* (<https://www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition>)—for sharing his thoughts with us as we were deciding how to approach this coroutines section.

18.3 Installing the `concurrentpp` and `generator` Libraries

`concurrentpp` Library

To install `concurrentpp`, follow the instructions for your platform at

<https://github.com/David-Haim/concurrentpp#building-installing-and-testing>

If you're using Visual C++, perform the following additional steps:

1. Open `concurrentpp.sln` from the library's `concurrentpp\build\lib` and build the solution to generate the library files.
2. In your Visual Studio solution that will use `concurrentpp`, follow the instructions presented in Section 3.14 to add the `concurrentpp\include` folder to the header include path.
3. Next, select **File > Add > Existing Project...**, navigate to the `concurrentpp` library's `concurrentpp\build\lib` folder and add `concurrentpp.vcxproj` to your solution.
4. Finally, right-click your project, select **Add > Reference...**, then in the **Add Reference** dialog, check the `concurrentpp` project and click **OK**.

`generator` Library

You can simply download the header-only `generator` library from

<https://github.com/TartanLlama/generator>

and include it in your project. If you prefer, you can clone the GitHub repository, then add the library's `include` folder to your compiler's header include path.

18.4 Creating a Generator Coroutine with `co_yield` and the `generator` Library

A `generator` is a coroutine that produces values on demand. When you call a generator, it uses a `co_yield` expression to suspend its execution and return the next generated value to its caller. Generator support is expected to be part of the C++23 standard library. For this example, we'll use the `generator` library. Its `t1::generator class template` enables you to specify the return type of a generator coroutine. It provides the mechanisms that enable `co_yield` to return values to a generator coroutine's caller.

Figure 18.1 defines a generator coroutine that produces the Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

which begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

```

1 // fig18_01.cpp
2 // Creating a generator coroutine with co_yield.
3 #include <format>
4 #include <iostream>
5 #include <sstream>
6 #include <thread>
7 #include <t1/generator.hpp>
8
9 // get current thread's ID as a string
10 std::string id() {
11     std::ostringstream out;
12     out << std::this_thread::get_id();
13     return out.str();
14 }
15
16 // coroutine that repeatedly yields the next Fibonacci value in sequence
17 t1::generator<int> fibonacciGenerator(int limit) {
18     std::cout << std::format(
19         "Thread {}: fibonacciGenerator started executing\n", id());
20
21     int value1{0}; // Fibonacci(0)
22     int value2{1}; // Fibonacci(1)
23
24     for (int i{0}; i < limit; ++i) {
25         co_yield value1; // yield current value of value1
26
27         // update value1 and value2 for next iteration
28         int temp{value1 + value2};
29         value1 = value2;
30         value2 = temp;
31     }
32
33     std::cout << std::format(
34         "Thread {}: fibonacciGenerator finished executing\n", id());
35 }
36
37 int main() {
38     std::cout << std::format("Thread {}: main begins\n", id());
39
40     // display first 10 Fibonacci values
41     for (int i{0}; auto value : fibonacciGenerator(10)) {
42         std::cout << std::format("Fibonacci({}) is {}\n", i++, value);
43     }
44
45     std::cout << std::format("Thread {}: main ends\n", id());
46 }
```

Thread 7316: main begins
 Thread 7316: fibonacciGenerator started executing
 Fibonacci(0) is 0
 Fibonacci(1) is 1
 Fibonacci(2) is 1

(continued...)

Fig. 18.1 | Creating a generator coroutine with `co_yield`. (Part 1 of 2.)

```
Fibonacci(3) is 2
Fibonacci(4) is 3
Fibonacci(5) is 5
Fibonacci(6) is 8
Fibonacci(7) is 13
Fibonacci(8) is 21
Fibonacci(9) is 34
Thread 7316: fibonacciGenerator finished executing
Thread 7316: main ends
```

Fig. 18.1 | Creating a generator coroutine with `co_yield`. (Part 2 of 2.)

`id` Function for Converting a Thread ID to a `std::string`

The `id` function (lines 10–14) uses a `std::thread::id` object’s overloaded `operator<<` to convert a thread ID to a `std::string`. We’ll use this to show that `main` and `fibonacciGenerator` execute in the same thread.

`fibonacciGenerator` Coroutine

Lines 17–35 define the `fibonacciGenerator` coroutine. The function’s `limit` parameter determines the number of Fibonacci values to produce, starting with `Fibonacci(0)`. Lines 18–19 display the thread ID in which `fibonacciGenerator` is executing. In the program’s output, you can see that the thread ID is the same as the one displayed by `main`, showing that the coroutine executes in the same thread as `main`. Lines 21 and 22 define variables `value1` and `value2`. Initially, these variables store the first two values in the Fibonacci sequence. When a program requests a new value from the generator, `co_yield` (line 25) suspends the coroutine’s execution and immediately returns the next Fibonacci value. When the code subsequently asks for the next value, the coroutine resumes and lines 28–30 update `value1` and `value2`. Then, the next iteration of the loop `co_yields` the next Fibonacci value, once again suspending the coroutine and returning a value to the caller. This continues until the last Fibonacci value is produced. Then, lines 33–34 again display the thread ID in which `fibonacciGenerator` is executing to show that it’s still the same thread as `main`.

`main` Function

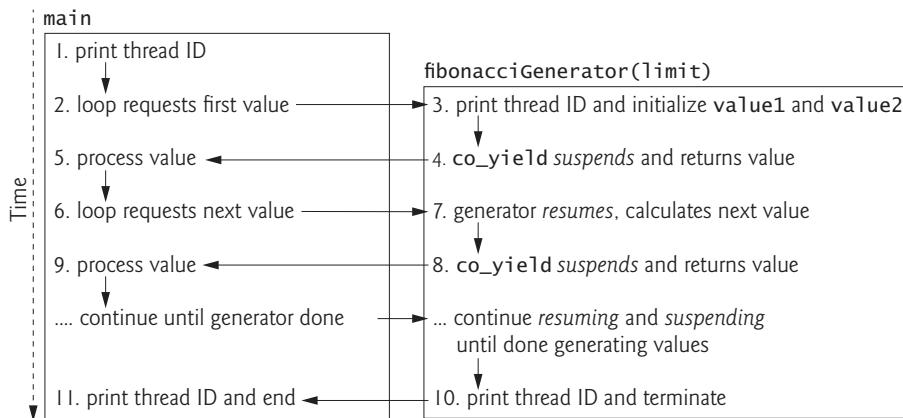
Line 38 displays `main`’s thread ID, so we can confirm that `main` and `fibonacciGenerator` execute in the same thread. Lines 41–43 request the `Fibonacci(0)` through `Fibonacci(9)` values. The call `fibonacciGenerator(10)` returns a `t1::generator<int>`, which provides iterators, so you can use it in a range-based `for` loop. When each iteration of the loop requests the next value, the `fibonacciGenerator` coroutine resumes execution to produce the next value, then suspends and returns it. Finally, line 45 displays `main`’s thread ID and shows that `main` ends. The thread IDs in the sample output confirm that `fibonacciGenerator` performs its work in `main`’s thread.

Calling a coroutine does not automatically create a new thread for you. If any threads are required, the coroutine must create them, as we’ll do in Sections 18.5 and 18.6. You can create and manage those threads using the techniques shown in Chapter 17, or you can let libraries like `concurrentpp` create and manage them for you.



Diagram Showing the Flow of Control for a Generator Coroutine

The following diagram shows the basic flow of control in this program—the numbers in this description correspond to the steps in the diagram:



1. `main` prints its thread ID.
2. The loop in `main` requests the first value from `fibonacciGenerator`.
3. `fibonacciGenerator` prints its thread ID and initializes its local variables.
4. `fibonacciGenerator` `co_yields` a value, which suspends the coroutine's execution and returns the value, and control, to `main`.
5. The loop in `main` processes the value.
6. The loop in `main` requests the next value from `fibonacciGenerator`.
7. `fibonacciGenerator` resumes execution and calculates the next value.
8. `fibonacciGenerator` `co_yields` a value. Again, this suspends the coroutine's execution and returns the value, and control, to `main`.
9. The loop in `main` processes the value.
10. Steps 6–9 continue until `fibonacciGenerator` finishes producing values. Then, `fibonacciGenerator` prints its thread ID, terminates and returns control to `main`.
11. The loop in `main` terminates, then `main` prints its thread ID and terminates.

Coroutines Are Stackless

The compiler manages the mechanisms that enable coroutines to suspend and resume. It creates the `coroutine state`,²² which contains the information required to resume a coroutine. This state is dynamically allocated on the heap rather than the stack, so coroutines are said to be `stackless`.²³ If the compiler determines that the coroutine's lifetime is entirely

22. C++ Standard, “Coroutine Definitions,” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/dc1.fct.def.coroutine#9>.

23. Varun Ramesh Blog, “Stackless vs. Stackful Coroutines,” August 18, 2017. Accessed April 18, 2023. <https://blog.varunramesh.net/posts/stackless-vs-stackful-coroutines/>.



within that of its caller, it can eliminate the heap allocation overhead.²⁴ Marc Gregoire points out that “memory usage for stackless coroutines is minimal, allowing for millions or even billions of coroutines to be running concurrently.”²⁵

18.5 Launching Tasks with `concurrency`

Before we use `concurrency` to implement a coroutine, let’s use it to **schedule tasks that execute in separate threads**. As with standard library threads, each task executes a function, a lambda or a function object. Figure 18.2 uses four `concurrency` components:

- A `concurrency::runtime` manages your `concurrency` interactions. It provides access to various executors for scheduling tasks. It also ensures that the executors shut down scheduled tasks properly when the `runtime` is destroyed. You must create a local `runtime` object, typically at the beginning of `main` (line 35). When `main` ends, the local `runtime` object goes out of scope. Its destructor shuts down the executors and any remaining scheduled tasks that have not finished executing.
- A `concurrency::thread_pool_executor` (one of several executor types) schedules tasks to execute. It creates and manages a group of threads called a `thread pool`²⁶ and assigns tasks to the pool’s threads to execute. This executor can **reuse existing threads in the pool to eliminate the overhead of creating a new thread for each task**. It also can **optimize the number of threads** to ensure the processor stays busy without creating so many threads that the application runs out of resources. Standard executors are expected in C++23 or C++26, but libraries like `concurrency`, `libunifex`²⁷ and `folly::coro` offer them now.
- A `concurrency::task` represents a task to execute. `concurrency` executors create these when you schedule tasks.
- A `concurrency::result` enables you to access a concurrent task’s result or simply wait for the task to complete execution if it does not return a value. When you schedule a `concurrency::task`, you receive a `concurrency::result`.



```
1 // fig18_02.cpp
2 // Setting up the concurrency::runtime and scheduling tasks with it.
3 #include <chrono>
4 #include <concurrency/concurrency.h>
5 #include <format>
6 #include <iostream>
7 #include <random>
8 #include <sstream>
9 #include <thread>
```

Fig. 18.2 | Setting up the `concurrency::runtime` and scheduling tasks with it. (Part 1 of 3.)

24. “Coroutines—Heap Allocation.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/coroutines#Heap_allocation.
25. Marc Gregoire, *Professional C++, Fifth Edition*, p. 963. 2021. Indianapolis, IN: John Wiley & Sons, 2021.
26. “Thread pool,” Accessed April 18, 2023. https://en.wikipedia.org/wiki/Thread_pool.
27. “libunifex.” Accessed April 18, 2023. <https://github.com/facebookexperimental/libunifex>.

```

10 #include <vector>
11
12 // get current thread's ID as a string
13 std::string id() {
14     std::ostringstream out;
15     out << std::this_thread::get_id();
16     return out.str();
17 }
18
19 // Function printTask sleeps for a specified period in milliseconds.
20 // When it continues executing, it prints its name and completes.
21 void printTask(std::string name, std::chrono::milliseconds sleep) {
22     std::cout << std::format(
23         "{} (thread ID: {}) going to sleep for {} ms\n",
24         name, id(), sleep.count());
25
26     // put the calling thread to sleep for sleep milliseconds
27     std::this_thread::sleep_for(sleep);
28
29     std::cout << std::format("{} (thread ID: {}) done sleeping\n",
30         name, id());
31 }
32
33 int main() {
34     // set up the concurrenccpp runtime for scheduling tasks to execute
35     concurrenccpp::runtime runtime;
36
37     std::cout << std::format("main's thread ID: {}\n\n", id());
38
39     // set up random number generation for random sleep times
40     std::random_device rd;
41     std::default_random_engine engine{rd()};
42     std::uniform_int_distribution ints{0, 5000};
43
44     // stores the tasks so we can wait for them to complete later;
45     // concurrenccpp::result<void> indicates that each task returns void
46     std::vector<concurrenccpp::result<void>> results;
47
48     std::cout << "STARTING THREE CONCURRENCCPP TASKS\n";
49
50     // schedule three tasks
51     for (int i{1}; i < 4; ++i) {
52         std::chrono::milliseconds sleepTime{ints(engine)};
53         std::string name{std::format("Task {}", i)};
54
55         // use a concurrenccpp thread_pool_executor to schedule a call
56         // to printTask with name and sleepTime as its arguments
57         results.push_back(runtime.thread_pool_executor()->submit(
58             printTask, name, sleepTime));
59     }
60
61     std::cout << "\nALL TASKS STARTED\n";
62     std::cout << "\nWAITING FOR TASKS TO COMPLETE\n";

```

Fig. 18.2 | Setting up the concurrenccpp::runtime and scheduling tasks with it. (Part 2 of 3.)

```
63      // wait for each task to complete
64      for (auto& result : results) {
65          result.get(); // wait for each task to return its result
66      }
67  }
68
69      std::cout << std::format("main's thread ID: {}\nMAIN ENDS\n", id());
70  }
```

```
main's thread ID: 20740
STARTING THREE CONCURRENCPPO TASKS
ALL TASKS STARTED

WAITING FOR TASKS TO COMPLETE
Task 3 (thread ID: 18960) going to sleep for 2683 ms
Task 1 (thread ID: 9840) going to sleep for 3856 ms
Task 2 (thread ID: 1700) going to sleep for 904 ms
Task 2 (thread ID: 1700) done sleeping
Task 3 (thread ID: 18960) done sleeping
Task 1 (thread ID: 9840) done sleeping

main's thread ID: 20740
MAIN ENDS
```

Fig. 18.2 | Setting up the `concurrentpp::runtime` and scheduling tasks with it. (Part 3 of 3.)

Function `id`

Function `id` (lines 13–17) creates a `std::string` representation of the current thread’s unique ID. We’ll use this to show the threads in which `main` and each of our tasks execute.

Function `printTask`

We do not create and manage the threads in this example. When you schedule tasks with `concurrentpp::thread_pool_executor`, it creates and manages threads for you using its **thread pool**. This example schedules calls to function `printTask` (lines 21–31) and shows that they execute on separate threads. Lines 22–24 and 29–30 display the currently executing thread’s unique ID to confirm that `printTask` is called from separate threads.

Function `main`

`main` is similar to the one in Fig. 17.4, so we’ll focus on the `concurrentpp` statements. Line 35 creates the `concurrentpp::runtime` so we can use the library’s features. Line 37 displays the thread ID in which `main` is executing.

concurrentpp Tasks and Results

Tasks allow developers to be more productive “by allowing them to focus more on business logic and less on low-level concepts like thread management and inter-thread synchronizations.”²⁸ You define your tasks as functions (or lambdas or function objects).

28. David Haim, “concurrentpp overview,” Accessed April 18, 2023. <https://github.com/David-Haim/concurrentpp#concurrentpp-overview>.

When you schedule a task, `concurrentpp` creates a `concurrentpp::task` object for you and returns a `concurrentpp::result` object representing the task's result, which might not be available until sometime in the future. If the task's function returns a result, you access it through the `concurrentpp::result` object, as we'll show in Fig. 18.3. If the function returns `void`, as `printTask` does, you can use the `concurrentpp::result<void>` object to wait for the task to complete, similar to joining a thread. Line 46 creates a vector of `concurrentpp::result<void>` objects. We use this to store the tasks' results, so we can wait for the tasks to complete later in `main`.

`concurrentpp::thread_pool_executor`

Programming languages like Java, Go, Python and Kotlin recommend implementing concurrency via an `executor` rather than creating and managing threads directly. This example uses a `concurrentpp::thread_pool_executor` (lines 57–58) to schedule each task. The `runtime` object's `thread_pool_executor` function returns a `shared_ptr`. All parts of your program use `shared_ptrs` to interact with a single object of a given `executor` type. The executor's `submit` function schedules a task to execute. It receives a variable number of arguments:

- `submit`'s first argument is the function defining the task to execute—in this case, `printTask`.
- `submit` passes its additional arguments to the function in its first argument—in this case, `submit` passes `name` and `sleepTime` to `printTask`.

Each `submit` call creates a `task` object and returns a `result`. Again, `printTask` returns `void`, so `submit` returns type `concurrentpp::result<void>` in this example.

Waiting for the Tasks to Complete Execution

Calling each `result` object's `get` function (lines 65–67) waits for the corresponding task's result—similar to joining a thread. Each task in this example returns `void`, so calling `get` simply causes `main` to wait for the task to complete. If `printTask` returned a value, `get` would return that value.

Summary of concurrentpp Executors

The `concurrentpp` documentation provides a list of executors and when to use each,²⁹ including:

- `thread_executor`—For each task, a `thread_executor` launches a separate thread that is not reused once it completes. According to the `concurrentpp` documentation, `thread_executor` “is good for long-running tasks, like objects that run a work loop, or long blocking operations.”
- `thread_pool_executor` (used in Figs. 18.2 and 18.3)—Recall that this schedules tasks using a pool of threads. “Suitable for short CPU-bound tasks that don't block. Applications are encouraged to use this executor as the default executor for non-blocking tasks.”

29. “`concurrentpp`—Executors.” Accessed April 18, 2023. <https://github.com/David-Haim/concurrentpp#executors>. Copyright 2020 David Haim.

- **background_executor**—“A thread pool executor with a larger pool of threads. Suitable for launching short blocking tasks like file I/O and DB [database] queries.”
- **worker_thread_executor**—“A single thread executor that maintains a single task queue. Suitable when applications want a dedicated thread that executes many related tasks.”
- **inline_executor** (discussed momentarily)—“Mainly used to override the behavior of other executors. Enqueuing a task is equivalent to invoking it inline.”

Tasks Are Not Required to Run on Separate Threads

A `concurrentpp::inline_executor` schedules tasks on the calling thread. To produce the following output, we replaced the `thread_pool_executor` (Fig. 18.2, line 57) with an `inline_executor`, which executes tasks on the calling thread (in this case, `main`). Every time you run this program with an `inline_executor`, the tasks execute sequentially in the same thread as `main` and in the order you schedule them. This sample output shows that all three tasks have the same thread ID as `main`. As we schedule each task, it immediately goes to sleep, then eventually wakes up and completes before the next task launches.

```
main's thread ID: 7854
STARTING THREE CONCURRENCPPP TASKS
Task 1 (ID: 7854) going to sleep for 2000 ms
Task 1 (ID: 7854) done sleeping
Task 2 (ID: 7854) going to sleep for 4432 ms
Task 2 (ID: 7854) done sleeping
Task 3 (ID: 7854) going to sleep for 3688 ms
Task 3 (ID: 7854) done sleeping

ALL TASKS STARTED

WAITING FOR TASKS TO COMPLETE

main's thread ID: 7854
MAIN ENDS
```

18.6 Creating a Coroutine with `co_await` and `co_return`

Now, let’s use `concurrentpp`, `co_await` and `co_return` to implement a coroutine that performs a potentially long-running task—sorting a 100-million-element vector of `int` values.

Overview of This Example

We’ll use `concurrentpp`’s `thread_pool_executor` to put two tasks to work in parallel, with each sorting half the vector. Once those tasks are complete, we’ll use the standard library algorithm `inplace_merge` (Section 14.4.9) to merge the vector’s two sorted halves. Figure 18.3 consists of the following functions:

- Function `id` (lines 14–18) converts a unique thread ID to a `string`.

- Coroutine `sortCoroutine` (lines 21–78) uses `concurrentpp` to launch two tasks that each sort half the vector.
- Function `main` (lines 80–107) creates the 100-million-element vector and calls `sortCoroutine` to sort the vector.

We've split this program into pieces for discussion purposes. After the program, we show a sample output.

#include Directives and Function id

In Fig. 18.3, lines 3–11 include the headers used in this program, and lines 14–18 define function `id`. The `concurrentpp` library (line 3) provides the coroutine support features this program requires.

```

1 // fig18_03.cpp
2 // Implementing a coroutine with co_await and co_return.
3 #include <concurrentpp/concurrentpp.h>
4 #include <format>
5 #include <iostream>
6 #include <memory> // for shared_ptr
7 #include <random>
8 #include <sstream>
9 #include <string>
10 #include <thread>
11 #include <vector>
12
13 // get current thread's ID as a string
14 std::string id() {
15     std::ostringstream out;
16     out << std::this_thread::get_id();
17     return out.str();
18 }
19

```

Fig. 18.3 | Implementing a coroutine with `co_await` and `co_return`.

sortCoroutine That Launches Two Tasks

Lines 21–78 define `sortCoroutine`, which receives as arguments

- a `std::shared_ptr<concurrentpp::thread_pool_executor>` used to schedule tasks and
- a reference to a `vector<int>` to sort.

```

20 // coroutine that sorts a vector<int> using two tasks
21 concurrentpp::result<void> sortCoroutine(
22     std::shared_ptr<concurrentpp::thread_pool_executor> executor,
23     std::vector<int>& values) {
24
25     std::cout << std::format("Thread {}: sortCoroutine started\n\n", id());
26

```

```
27 // lambda that sorts a portion of a vector
28 auto sortTask{
29     [&](auto begin, auto end) {
30         std::cout << std::format(
31             "Thread {}: Sorting {} elements\n", id(), end - begin);
32         std::sort(begin, end);
33         std::cout << std::format("Thread {}: Finished sorting\n", id());
34     }
35 };
36
37 // stores task results
38 std::vector<concurrentpp::result<void>> results;
39
40 size_t middle{values.size() / 2}; // middle element index
41
42 std::cout << std::format(
43     "Thread {}: sortCoroutine starting first half sortTask\n", id());
44
45 // use a concurrentpp thread_pool_executor to schedule
46 // a sortTask call that sorts the first half of values
47 results.push_back(
48     executor->submit(
49         [&]() {sortTask(values.begin(), values.begin() + middle);}
50     )
51 );
52
53 std::cout << std::format(
54     "Thread {}: sortCoroutine starting second half sortTask\n", id());
55
56 // use a concurrentpp thread_pool_executor to schedule
57 // a sortTask call that sorts the second half of values
58 results.push_back(
59     executor->submit(
60         [&]() {sortTask(values.begin() + middle, values.end());}
61     )
62 );
63
64 // suspend coroutine while waiting for all sortTasks to complete
65 std::cout << std::format("\nThread {}: {}\n", id(),
66                         "sortCoroutine co_awaiting sortTask completion");
67 co_await concurrentpp::when_all(
68     executor, results.begin(), results.end());
69
70 // merge the two sorted sub-vectors
71 std::cout << std::format(
72     "\nThread {}: sortCoroutine merging results\n", id());
73 std::inplace_merge(
74     values.begin(), values.begin() + middle, values.end());
75
76 std::cout << std::format("Thread {}: sortCoroutine done\n", id());
77 co_return; // terminate coroutine and resume caller
78 }
79 
```

The `sortCoroutine` operates as follows:

- Line 25 displays a message containing the thread ID in which `sortCoroutine` is running and indicating that `sortCoroutine` started. The thread ID confirms that `sortCoroutine` runs in the same thread as `main`.
- The `sortCoroutine` will launch two `concurrentpp` tasks that execute the lambda `sortTask` (lines 28–35). The lambda receives random-access iterators representing the beginning and end of a common range to sort. Lines 30–31 display a message containing the thread ID in which a given call to `sortTask` executes and the number of elements it is sorting. Line 32 calls the `std::sort` algorithm to sort the specified portion of the `vector`. When the sort completes, line 33 displays a message containing the thread ID and indicating that the task's sort completed.
- The `sortTask` does not return a value, so each `thread_pool_executor`'s `submit` function will return a `concurrentpp::result<void>` for each task. We'll store these in the `vector results` (defined in line 38).
- Line 40 determines the `vector`'s middle element, which we'll use to divide the `vector` in half.
- Lines 42–43 display a message indicating the executing thread's ID and that we're starting the `sortTask` for the first half of the `vector`. Then lines 47–51 use `thread_pool_executor`'s `submit` function to create a task that calls `sortTask` to sort the elements in the range `values.begin()` up to, but not including, the middle `values` element. We store the `result` object returned by `submit` in the `results` vector. Lines 53–62 repeat these steps to launch a second task that sorts the `values` elements from the `middle` element through the end of the `vector`.
- Lines 65–66 display a message containing the thread ID in which `sortCoroutine` is running and indicate that we are `co_awaiting` the `sortTask` results. Lines 67–68 use `concurrentpp`'s `when_all` function to `co_await` all the `sortTasks`' results. We'll discuss this line in more detail momentarily.
- Lines 71–72 display a message containing the thread ID in which `sortCoroutine` is running and indicate that we are merging the results. Then, lines 73–74 use the standard library algorithm `inplace_merge` to merge the `vector`'s two sorted halves.
- Finally, line 76 displays the thread ID of the thread in which `isPrime` is currently executing and indicates that `sortCoroutine` is done. Then, line 77 uses the C++20 `co_return statement` to terminate the coroutine and return control to `main`. Because `sortCoroutine` does not return a value, `concurrentpp` returns a `concurrentpp::result<void>` to the coroutine's caller. You'll see that `main` uses this object's `get` function to wait for the coroutine to finish executing.

Using `when_all` with C++20's `co_await` Expression

A coroutine utility function we might see in C++23 or later is `when_all`, which enables a program to wait for a set of tasks to complete. Function `when_all` receives as arguments



- an executor that's used to resume the coroutine's execution once all the tasks are complete and
- iterators representing a common range of `concurrentpp::result` objects—in this case, all the elements of the `results` vector.

Lines 67–68 `co_await` the result of the `when_all` call, enabling `main` to continue its work while the sorting occurs. A `co_await expression` consists of the `co_await operator` followed by an expression that returns an `awaitable entity`. Typically, this will be an object of a coroutine-library class. A `concurrentpp::result` satisfies the C++ standard's requirements for the operand of the `co_await operator`, as does the object `when_all` returns, which is a `concurrentpp::lazy_result` object containing a `tuple` of all the tasks' results.

A SE

Determining Whether to Suspend Coroutine Execution

At this point, the coroutine determines whether to `suspend` its execution. Before suspending, the coroutine calls the `co_await` operand's `await_ready` function to check whether a task's result is available. If so, both `sortTasks` already completed, so `sortCoroutine` continues executing with the next statement in sequence. Otherwise, the `sortCoroutine` coroutine suspends execution until the asynchronous tasks in `when_all`'s arguments complete execution. This allows the caller (`main`) to perform other work that does not depend on the results of the asynchronous tasks.

As you'll see in the sample output, if the coroutine suspends, `main` shows that it's executing again by displaying a line of text. Then, `main` waits for `sortCoroutine` to complete and return. However, rather than waiting for `sortCoroutine`, `main` could continue doing other work.

Resuming Coroutine Execution

When the `co_awaited` asynchronous tasks are complete, the `sortCoroutine` resumes execution and continues with the next statement after the `co_await expression`. In this example, `sortCoroutine` merges the sorted first half and sorted second half of the vector, then `co_returns` control to `main`, so it can continue executing.

`concurrentpp when_any Utility Function`

Another coroutine utility function we might see in C++23 or later is `when_any`, which enables a program to wait for any of two or more tasks to complete. `concurrentpp` provides a `when_any` function.

One `when_any` use-case might be downloading several large files—one per task. Though you might want all the results eventually, you'd like to start processing once the first download is complete. You could then call `when_any` again for the remaining tasks that are still executing. `concurrentpp`'s `when_any` typically is used in a loop that keeps iterating until the last of several tasks completes.

A SE

`main` Program

Function `main` (lines 80–107) creates a vector containing 100 million random `int` values, calls `sortCoroutine` to sort the vector, waits for `sortCoroutine` to complete, then confirms that the vector is sorted.

```

80 int main() {
81     concurrency::runtime runtime; // set up concurrency::runtime
82     auto executor{runtime.thread_pool_executor()}; // get the executor
83
84     // set up random number generation
85     std::random_device rd;
86     std::default_random_engine engine{rd()};
87     std::uniform_int_distribution ints;
88
89     std::cout << std::format(
90         "Thread {}: main creating vector of random ints\n", id());
91     std::vector<int> values(100'000'000);
92     std::ranges::generate(values, [&] () {return ints(engine);});
93
94     std::cout << std::format(
95         "Thread {}: main starting sortCoroutine\n", id());
96     auto result{sortCoroutine(executor, values)};
97
98     std::cout << std::format("\nThread {}: {}\n", id(),
99         "main resumed. Waiting for sortCoroutine to complete.");
100    result.get(); // wait for sortCoroutine to complete
101
102    std::cout << std::format(
103        "\nThread {}: main confirming that vector is sorted\n", id());
104    bool sorted{std::ranges::is_sorted(values)};
105    std::cout << std::format("Thread {}: values is{} sorted\n",
106        id(), sorted ? "" : " not");
107 }

```

The `main` function operates as follows:

- Lines 81–82 set up the `concurrency::runtime` object and get its `concurrency::thread_pool_executor`.
- Lines 85–87 set up the random-number generation to populate a `vector<int>`.
- Lines 89–92 create the 100-million-element `vector<int>`, then fill it with random `int` values using the `std::ranges::generate` algorithm.
- Lines 94–95 display that the `main` thread is executing and about to call `sortCoroutine`.
- Line 96 calls coroutine `sortCoroutine`, passing the `executor` and the `vector values`. This launches the asynchronous sorting tasks and returns an object of type `concurrency::result<void>`. The `sortCoroutine` call executes in `main`'s thread to launch the asynchronous tasks. When `sortCoroutine` `co_awaits` those tasks, it will suspend its execution if their results are not yet available; otherwise, it will simply complete and return.
- In this example, the asynchronous tasks will not be complete because it takes time for each task launched by `sortCoroutine` to sort 50 million elements. So, the coroutine will suspend its execution and return control to `main`. At this point, lines 98–99 will show that `main` is executing again by displaying a line of text. This is where `main` could potentially continue doing other work.

- Line 100 calls the `result` object's `get` method to block `main` from continuing until `sortCoroutine`'s asynchronous tasks finish executing.
- Once that happens, lines 102–103 display a message indicating that `main` is confirming values is sorted. Then, line 104 calls `std::ranges::is_sorted` with the `values` as an argument. This algorithm returns `true` if its argument is sorted; otherwise, it returns `false`. Finally, lines 105–106 display a message indicating whether the vector is sorted.

Sample Output

Let's discuss the sample output in detail. Below the output, we broke it into pieces for discussion purposes.

```
Thread 17276: main creating vector of random ints
Thread 17276: main starting sortCoroutine
Thread 17276: sortCoroutine started

Thread 17276: sortCoroutine starting first half sortTask
Thread 17276: sortCoroutine starting second half sortTask

Thread 17276: sortCoroutine co_awaiting sortTask completion

Thread 17276: main resumed. Waiting for sortCoroutine to complete.
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
Thread 6252: Finished sorting
Thread 17144: Finished sorting

Thread 6252: sortCoroutine merging results
Thread 6252: sortCoroutine done

Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

First, `main` displays its thread ID and creates the vector of random integers:

```
Thread 17276: main creating vector of random ints
```

Next, `main` displays its thread ID and calls `sortCoroutine`:

```
Thread 17276: main starting sortCoroutine
```

Then, `sortCoroutine` displays its thread ID and indicates that it started:

```
Thread 17276: sortCoroutine started
```

Note that `sortCoroutine` executes in `main`'s thread, so it displays the same thread ID.

Next, `sortCoroutine` launches two tasks to sort the vector:

```
Thread 17276: sortCoroutine starting first half sortTask
```

```
Thread 17276: sortCoroutine starting second half sortTask
```

At this point, `sortCoroutine` co_awaits completion of the tasks:

```
Thread 17276: sortCoroutine co_awaiting sortTask completion
```

If the sortTasks have not completed, `sortCoroutine` suspends execution and returns control to `main`. We purposely created a large vector so the sorting would not complete immediately. This enables us to show that `main` continues executing when the coroutine suspends execution:

```
Thread 17276: main resumed. Waiting for sortCoroutine to complete.
```

`main` displays the preceding line of text, then calls the `sortCoroutine` object's `get` function to block until `sortCoroutine` completes and returns control to `main`.

In the meantime, the parallel `sortTasks` start running in other threads, as shown by their thread IDs:

```
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
```

We cannot predict the relative speeds of asynchronous concurrent tasks, so we do not know which `sortTask` will finish first. In this run, the `sortTasks` finished in the reverse of the order they started, as confirmed by their thread IDs:

```
Thread 6252: Finished sorting
Thread 17144: Finished sorting
```

Next, `sortCoroutine` merges the results of the two `sortTasks`, then indicates that it's done:

```
Thread 6252: sortCoroutine merging results
Thread 6252: sortCoroutine done
```

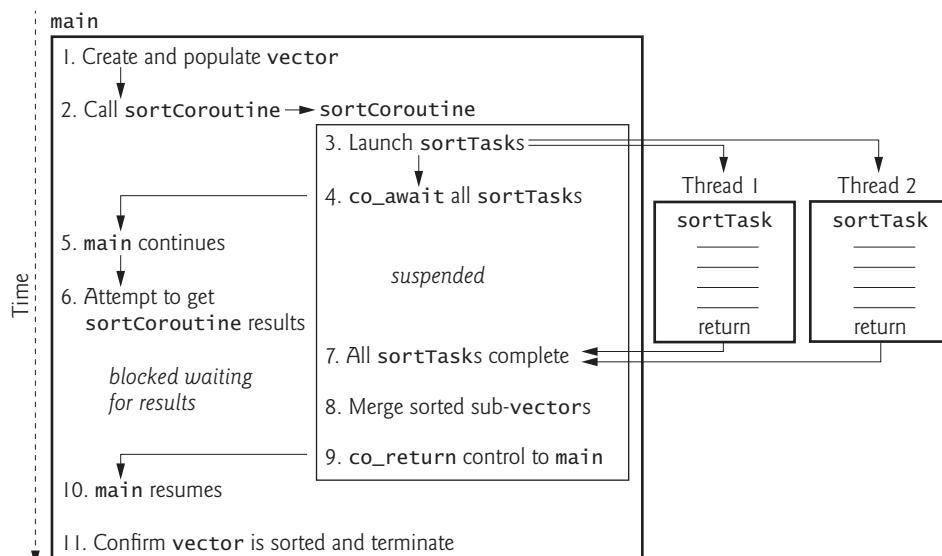
The executor resumed `sortCoroutine` on the thread of the last `sortTask` to complete.

At this point, `sortCoroutine co_returns` control to `main`, which continues executing, confirms that the vector is sorted, then terminates:

```
Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

Flow of Control for a Coroutine with `co_await` and `co_return`

The following diagram shows the basic flow of control in this program—the numbers in this description correspond to the steps in the diagram:



1. `main` creates and populates the vector of random integers.
2. `main` calls `sortCoroutine`.
3. `sortCoroutine` launches two parallel tasks that each call `sortTask` for half the vector. Those tasks begin running in parallel.
4. `sortCoroutine` co_awaits all of the tasks' results, suspending execution of the coroutine and returning control to `main`.
5. `main` continues executing.
6. `main` attempts to get `sortCoroutine`'s result. This blocks `main`'s from continuing until `sortCoroutine` completes execution and returns control to `main`.
7. Eventually, the `sortTasks` complete, at which point `sortCoroutine` resumes execution.
8. `sortCoroutine` merges the sorted sub-vectors.
9. `sortCoroutine` co_returns control to `main`, terminating `sortCoroutine`.
10. `main` resumes execution.
11. `main` confirms the vector is sorted and terminates.

18.7 Low-Level Coroutines Concepts

You've now seen that coroutine support libraries, like `generator` and `concurrentpp`, make it convenient to create coroutines. For programmers who want to try using coroutines without using these "experimental" coroutine support libraries, this section overviews some of the low-level concepts. For further study, we've included key articles and videos with code examples in the footnotes.^{30,31,32,33,34,35}

This section also will strengthen your understanding of the high-level, library-based approach we've taken in the preceding subsections. The details for topics discussed in this section can be found in the C++ standard's sections 7.6.2.3, 7.6.17, 8.7.4, 9.5.4 and 17.12,³⁶ and at cppreference.com.^{37,38}

-
30. Marius Bancila, *Modern C++ Programming Cookbook*, Chapter 12, pp. 681–700. Packet, 2020.
 31. Simon Toth, "C++20 Coroutines: Complete Guide," September 26, 2021. Accessed April 18, 2023. <https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d>.
 32. Rainer Grimm, "40 Years of Evolution from Functions to Coroutines," September 25, 2020. Accessed April 18, 2023. <https://www.youtube.com/watch?v=jd6P9X812bY>.
 33. Jeff Thomas, "Exploring Coroutines," April 7, 2021. Accessed April 18, 2023. <https://blog.cofeetocode.com/2021/04/exploring-coroutines/>.
 34. Panicsoftware, "Your First Coroutine," March 6, 2019. Accessed April 18, 2023. <https://blog.panicsoftware.com/your-first-coroutine/>.
 35. Panicsoftware, "co_awaiting Coroutines," April 12, 2019. Accessed April 18, 2023. https://blog.panicsoftware.com/co_awaiting-coroutines/.
 36. "C++ Standard," Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/>.
 37. "Coroutines (C++20)." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/coroutines>.
 38. "Standard Library Header <coroutine>." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/header/coroutine>.

Coroutine Restrictions

According to the C++20 standard, the following kinds of functions cannot be coroutines:³⁹

- `main`,
- constructors,
- destructors,
- `constexpr` and `consteval` functions,
- functions declared with placeholder types (`auto` or `concepts`) and
- functions with variable numbers of arguments.

Promise Object

Every coroutine has a **promise object**⁴⁰ that's created when the coroutine is first called. It's used by the coroutine to

- return a result to its caller or
- return an exception to its caller.

The promise object is part of the coroutine state. It provides several member functions:

- **get_return_object**—When the coroutine begins executing, it calls this function on the **promise object** to get the coroutine's **result object**, which will be returned to the coroutine's caller when the coroutine first **suspends**. The caller uses the **result object** to access the coroutine's result when it becomes available.
- **initial_suspend**—When the coroutine begins executing, it calls this function on the **promise object** to determine whether to immediately **suspend** (for **lazy coroutines**, like our `fibonacciGenerator` in Fig. 18.1) or **continue executing** (like our `sortCoroutine` coroutine in Fig. 18.3). The `<coroutine>` header provides types **suspend_always** and **suspend_never** to support these two possibilities.
- **unhandled_exception**—The coroutine calls this function on the **promise object** if an **unhandled exception** causes the coroutine to terminate.
- **return_void**—The coroutine calls this function on the **promise object** when execution reaches the coroutine's closing brace, a `co_return` statement with no expression or a `co_return` statement with an expression that evaluates to `void`. The **promise object** may define only **return_void** or **return_value**, but not both.
- **return_value**—The coroutine calls this function when a `co_return` statement contains a **non-void expression**. The **promise object** may define either **return_value** or **return_void**, but not both.

39. C++ Standard, Sections 6.9.3.1, 9.2.5, 9.2.8.5, 11.4.4, 11.4.6. Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/>.

40. Lewis Baker, "C++ Coroutines: Understanding the Promise Type," September 5, 2018. Accessed April 18, 2023. <https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type>.

- **final_suspend**—After calling `return_void` or `return_value`, the coroutine calls this function to return control, and possibly a result, to the coroutine’s caller.
- **yield_value**—When a `co_yield` statement executes, a generator coroutine calls this function to return a value to the coroutine’s caller.

Coroutine State

The **coroutine state** (sometimes called the **coroutine frame**) maintains

- copies of the coroutine’s **parameters**,
- its **promise object**,
- a representation of the **suspension point**, so the coroutine knows where to **resume execution**, and
- local variables in scope at the **suspension point**.

In our **generator** example (Fig. 18.1), the **suspension point** is where the `co_yield` statement suspends the coroutine and returns a value to the generator’s caller. In our sorting example (Fig. 18.3), the **suspension point** is where we `co_await` the result of calling `concurrentpp::when_all`.

Coroutine Handle

Generator support libraries use a **coroutine_handle** (from header `<coroutine>`) to refer to and manipulate a **running** or **suspended** coroutine. Some capabilities of class template **coroutine_handle** include:

- creating a **coroutine_handle** from a **promise object**,
- checking whether the referenced coroutine has **completed execution**,
- **resuming a suspended coroutine**,
- **destroying the coroutine state** and
- getting the coroutine’s **promise object**.

Awaitable Objects

The operand of `co_await` must be an **awaitable object**, such as a `concurrentpp::result` (Fig. 18.3). Such objects must provide several member functions:

- **await_ready**—When you `co_await` an awaitable object, this function is called first to determine whether the result is already available. If so, the coroutine continues executing.
- **await_suspend**—If the result is not ready, this function is called to determine whether to suspend the coroutine’s execution and return control to the caller. If this function returns a **coroutine_handle** for another suspended coroutine, that coroutine resumes execution.
- **await_resume**—When the coroutine resumes execution, this function is called to return the result of the `co_await` expression.

18.8 Future Coroutines Enhancements

Section 18.5 demonstrated **concurrentpp** executors for launching tasks and managing threads. **Standard library executors** and other features to support coroutine development are expected in C++23 and later. Various other third-party libraries have experimental executor implementations. For example, Facebook’s **libunifex** (“**unified executors**”) library⁴¹ is a prototype implementation of **executors** and other asynchronous programming facilities currently being considered for standardization. For more information, see the C++ Standards Committee proposals:

- “`std::execution`,”⁴²
- “A Unified Executors Proposal for C++”⁴³ and
- “Dependent Execution for a Unified Executors Proposal for C++.”⁴⁴

18.9 Wrap-Up

This chapter presented a detailed higher-level treatment of coroutines—the last of C++20’s “big four” features (ranges, concepts, modules and coroutines). The limited C++20 library support for coroutines is meant for developers creating higher-level coroutines support libraries for C++23 and later. We used this high-level, library-based approach to conveniently create coroutines, enabling sophisticated concurrent programming with a simple sequential-like coding style. We used the `co_yield` keyword and the `generator` library to implement a lazy generator coroutine. We also introduced the `concurrentpp` library’s executors, tasks and results, using them to execute tasks and define a coroutine with the `co_await` and `co_return` keywords. In the next chapter, we’ll discuss C++ stream I/O and C++20 text formatting in more depth.

Self-Review Exercises

18.1 Fill in the blanks in each of the following statements:

- a) Typically, a program would provide an asynchronous task with a(n) _____ (a function, a lambda or a function object) to call when the task completes.
- b) A(n) _____ is a function that can suspend execution and be resumed later.
- c) If a program does not need all the values at once, _____ can reduce your program’s memory consumption and improve performance.
- d) Every coroutine has a _____ that’s created when the coroutine is first called. It’s used by the coroutine to return a result to its caller or return an exception to its caller.

41. “libunifex.” Accessed April 18, 2023. <https://github.com/facebookexperimental/libunifex>.

42. Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler and Bryce Adelstein Lelbach, “`std::execution`,” October 4, 2021. Accessed April 18, 2023. <https://wg21.link/p2300>.

43. Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Daisy Hollman, Lee Howes, Kirk Shoop, Lewis Baker and Eric Niebler, “A Unified Executors Proposal for C++,” September 15, 2020. Accessed April 18, 2023. <http://wg21.link/p0443>.

44. Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards and Gordon Brown, “Dependent Execution for a Unified Executors Proposal for C++,” October 8, 2018. Accessed April 18, 2023. <https://wg21.link/p1244>.

- e) When a coroutine begins executing, it calls the _____ function on the promise object to determine whether to immediately suspend (for lazy coroutines) or continue executing.
 - f) Generator support libraries use a _____ (from header `<coroutine>`) to refer to and manipulate a running or suspended coroutine. Some capabilities of this class template include: creating a `coroutine_handle` from a promise object, checking whether the referenced coroutine has completed execution, resuming a suspended coroutine, destroying the coroutine state and getting the coroutine's promise object.
 - g) The operand of `co_await` must be a(n) _____ object, such as a `concurrentpp::result`.
 - h) If the result is not ready, the _____ function is called to determine whether to suspend the coroutine's execution and return control to the caller. If this function returns a `coroutine_handle` for another suspended coroutine, that coroutine resumes execution.
- 18.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- a) The mechanisms that suspend a coroutine and return control to its caller and continue a suspended coroutine's execution later are handled entirely by code that's written for you by the compiler.
 - b) A function containing any of the keywords `await`, `yield` or `return` is a coroutine.
 - c) A generator is a coroutine that produces values on demand.
 - d) Memory usage for stackless coroutines is substantial, severely limiting the number of coroutines that can run concurrently.
 - e) Tasks allow developers to be more productive by focusing less on business logic and more on low-level concepts like thread management and inter-thread synchronizations.
 - f) We can use `concurrentpp`'s `thread_pool_executor` to put multiple tasks to work in parallel.
 - g) When a coroutine `co_awaits` tasks, it will suspend its execution if their results are not yet available; otherwise, it will simply complete and return.
 - h) According to the C++20 standard, the following kinds of functions cannot be coroutines: `main`, constructors, destructors, `constexpr` and `consteval` functions, functions declared with placeholder types (auto or C++20 concepts) and functions with variable numbers of arguments.

Answers to Self-Review Exercises

18.1 a) callback function. b) coroutine. c) generators. d) promise object. e) `initial_suspend`. f) `coroutine_handle`. g) awaitable. h) `await_suspend`.

18.2 a) True. b) False. Actually, a function containing any of the keywords `co_await`, `co_yield` or `co_return` is a coroutine. c) True. d) False. Actually, Marc Gregoire points out that "memory usage for stackless coroutines is minimal, allowing for millions or even billions of coroutines to be running concurrently." e) False. Actually, tasks allow developers to be more productive by focusing more on business logic and less on low-level concepts like thread management and inter-thread synchronizations. f) True. g) True. h) True.

Exercises

- 18.3** Fill in the blanks in each of the following statements:
- When a program contains a long-running task, it's common for a function that you call _____—that is, performing tasks one after another—to launch the long-running task asynchronously.
 - Coroutines use cases include lazily computed sequences known as _____ that produce one value at a time on demand.
 - When you call a generator, it uses a(n) _____ expression to suspend its execution and return the next generated value to its caller.
 - The compiler manages the mechanisms that enable coroutines to suspend and resume. It creates the coroutine state, which contains the information required to resume a coroutine. This state is dynamically allocated on the heap rather than the stack, so coroutines are said to be _____.
 - When you schedule a `concurrentpp::task`, you receive a(n) _____.
 - The coroutine state (sometimes called the coroutine frame) maintains its _____ object, copies of the coroutine's parameters, a representation of the suspension point, so the coroutine knows where to resume execution, and local variables in scope at the suspension point.
 - When you `co_await` an awaitable object, the _____ function is called first to determine whether the result is already available. If so, the coroutine continues executing.
 - When the coroutine resumes execution, this function is called to return the result of the `co_await` expression _____.
- 18.4** State whether each of the following is *true* or *false*. If *false*, explain why.
- Coroutines enable you to do concurrent programming with a simple parallel coding style.
 - Coroutines require various supporting classes with numerous member functions and nested types. Creating these yourself is straightforward.
 - Calling a coroutine creates a new thread for you.
 - When you schedule tasks with `concurrentpp::thread_pool_executor`, it creates and manages threads for you using its thread pool.
 - A `background_executor` is “a thread pool executor with a larger pool of threads. Suitable for launching short blocking tasks like file I/O and DB [database] queries.”
 - A `worker_thread_executor` is a single-thread executor that maintains a single task queue. Suitable when applications want a dedicated thread that executes many related tasks.”
 - We cannot predict the relative speeds of asynchronous concurrent tasks.
 - The operand of `co_await` must be an awaitable object, such as a `concurrentpp::result`.

Discussion Exercises

- 18.5** (*Discuss `concurrentpp::runtime`*) Discuss how A `concurrentpp::runtime` objct manages your `concurrentpp` interactions.

18.6 (*Discuss `concurrentpp::thread_pool_executor`*) Describe how a `concurrentpp::thread_pool_executor` (one of several executor types) schedules tasks to execute.

Code Exercises

18.7 (*Generating Integer Ranges with Coroutines*) Using Fig. 18.1 as a guide, create a generator coroutine named `range` that produces an integer range of values. Provide three parameters:

- `start` represents the starting integer in the range.
- `end` represents the ending integer in the range—your generator coroutine should produce values up to, but not including, this value.
- `step` represents the increment between integers and has a default argument value of 1.

So calling `range(0, 10)` would generate the integers 0–9, calling `range(0, 10, 2)` would produce the even integers 0, 2, 4, 6 and 8, and calling `range(10, 0, -2)` would produce the even integers 10, 8, 6, 4 and 2. Thoroughly test your new generator function.

18.8 (*Generating Tuples with Coroutines*) Using Fig. 18.1 as a guide, create a generator coroutine named `zip` that receives two ranges of the same length and returns `std::tuple` objects containing the corresponding elements from each range. For example, given the collections of data

Susan, Eduardo, Pantipa

and

92, 83, 97

the `zip` generator would produce tuples containing

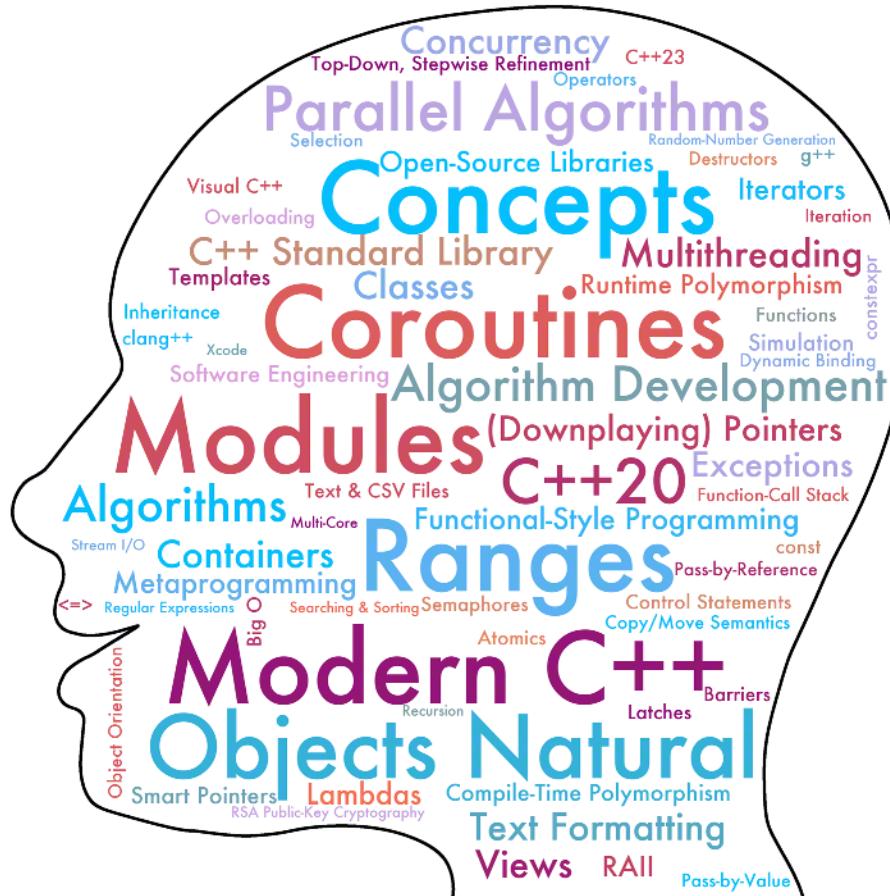
Susan, 92
Eduardo, 83
Pantipa, 97

18.9 (*Processing Asynchronous Tasks in Separate Threads with Coroutines*) Using the programs of Sections 18.5–18.6 as your guide, implement prime-factorization program of Section 17.10 using coroutines.

This page intentionally left blank

Stream I/O & C++20 Text Formatting

19



Objectives

In this chapter, you'll:

- Understand C++ object-oriented stream input/output used in legacy applications.
 - Input and output individual characters.
 - Use unformatted I/O for high performance.
 - Use stream manipulators to display integers in octal and hexadecimal formats.
 - Specify precision for input and output.
 - Display floating-point values in scientific and fixed-point notation.
 - Set and restore the format state.
 - Control alignment and padding.
 - Determine the success or failure of input/output operations.
 - Tie output streams to input streams.
 - See many of C++20's concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.

Outline

19.1	Introduction	
19.2	Streams	
19.2.1	Classic Streams vs. Standard Streams	
19.2.2	<code>iostream</code> Library Headers	
19.2.3	Stream Input/Output Classes and Objects	
19.3	Stream Output	
19.3.1	Output of <code>char*</code> Variables	
19.3.2	Character Output Using Member Function <code>put</code>	
19.4	Stream Input	
19.4.1	<code>get</code> and <code>getline</code> Member Functions	
19.4.2	<code>istream</code> Member Functions <code>peek</code> , <code>putback</code> and <code>ignore</code>	
19.5	Unformatted I/O Using <code>read</code> , <code>write</code> and <code>gcount</code>	
19.6	Stream Manipulators	
19.6.1	Integral Stream Base <code>dec</code> , <code>oct</code> , <code>hex</code> and <code>setbase</code>	
19.6.2	Floating-Point Precision (<code>setprecision</code> , <code>precision</code>)	
19.6.3	Field Width (<code>width</code> , <code>setw</code>)	
19.6.4	User-Defined Output Stream Manipulators	
19.6.5	Trailing Zeros and Decimal Points (<code>showpoint</code>)	
19.6.6	Alignment (<code>left</code> , <code>right</code> and <code>internal</code>)	
	19.6.7 Padding (<code>fill</code> , <code>setfill</code>)	
	19.6.8 Integral Stream Base (<code>dec</code> , <code>oct</code> , <code>hex</code> , <code>showbase</code>)	
	19.6.9 Floating-Point Numbers; Scientific and Fixed Notation (<code>scientific</code> , <code>fixed</code>)	
	19.6.10 Uppercase/Lowercase Control (<code>uppercase</code>)	
	19.6.11 Specifying Boolean Format (<code>boolalpha</code>)	
	19.6.12 Setting and Resetting the Format State via Member Function <code>flags</code>	
19.7	Stream Error States	
19.8	Tying an Output Stream to an Input Stream	
19.9	C++20 Text Formatting	
19.9.1	C++20 <code>std::format</code> Presentation Types	
19.9.2	C++20 <code>std::format</code> Field Widths and Alignment	
19.9.3	C++20 <code>std::format</code> Numeric Formatting	
19.9.4	C++20 <code>std::format</code> Field Width and Precision Placeholders	
19.10	Wrap-Up	
	Exercises	

19.1 Introduction

This chapter discusses input/output formatting capabilities. First, we present the I/O stream formatting capabilities that you’re likely to see in C++ legacy code. Then, we discuss features from C++20’s new text-formatting capabilities. In Section 19.9, you’ll see that C++20 text formatting is more concise and convenient than the I/O stream formatting capabilities presented in Section 19.6.

C++ uses **type-safe I/O**. Each I/O operation is executed in a manner sensitive to the data type. If an I/O function has been defined to handle a particular data type, then that function is called to handle that data type. If there is no match between the type of the actual data and a function for handling data of that type, the compiler generates an error. As you saw in Chapter 11, you can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (`<<`) and the stream extraction operator (`>>`).

19.2 Streams

C++ I/O occurs in **streams**, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection).

An application associates meaning with bytes. The bytes could represent characters, raw data, graphics images, audio, video or other information an application requires. The system

I/O mechanisms transfer bytes from devices to memory and vice versa. The time these transfers typically take is far greater than the time the processor requires to manipulate data in memory. I/O operations require careful planning and tuning to ensure optimal performance.

C++ provides both “low-level” and “high-level” I/O capabilities. Low-level **unformatted I/O** capabilities specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient. Higher-level **formatted I/O** groups bytes into meaningful units, such as integers, floating-point numbers, characters, strings and custom types.



19.2.1 Classic Streams vs. Standard Streams

C++’s **classic stream libraries** originally supported only `char`-based I/O. Because a `char` occupies one byte, it can represent only a limited set of characters, such as those in the ASCII character set. Many languages use alphabets that contain more characters than a single-byte `char` can represent. Such characters are typically available in the extensive international **Unicode® character set** (<https://unicode.org>), which can represent most of the world’s languages, mathematical symbols, emoji characters and more. C++ supports Unicode via the types

- `wchar_t`,
- `char16_t` and `char32_t`, and
- `char8_t` (C++20).

The **standard stream library classes** are class templates that can be instantiated for these various character types. We use the predefined stream-library instantiations for type `char` in this book. Unicode-based applications would use appropriate class-template instantiations based on the preceding types. C++ also supports Unicode string literals—for more information, see

https://en.cppreference.com/w/cpp/language/string_literal

19.2.2 `iostream` Library Headers

The C++ stream libraries provide hundreds of I/O capabilities. Most of our C++ programs include the `<iostream>` header, which declares basic services required for all stream-I/O operations. The `<iostream>` header defines the `cin`, `cout`, `cerr` and `clog` objects, which correspond to the **standard input stream**, the **standard output stream**, the **unbuffered standard error stream** and the **buffered standard error stream**, respectively. The next section discusses `cerr`, `clog` and buffering. The `<iostream>` and `<iomanip>` headers define **stream manipulators** for formatted I/O. We’ll demonstrate many of these in this chapter. We’ll also show that the newer C++20 text formatting with the `format` function greatly simplifies formatting output.

19.2.3 Stream Input/Output Classes and Objects

This chapter focuses on the `iostream` class templates:

- `basic_istream` for stream input operations and
- `basic_ostream` for stream output operations.

Though we do not use it in this chapter, `basic_iostream` provides stream input and stream output operations.

For each of the class templates `basic_istream`, `basic_ostream` and `basic_iostream`, the `iostream` library defines a type alias for char-based I/O:

- `istream` is a `basic_istream<char>` for char input—this is `cin`'s type.
- `ostream` is a `basic_ostream<char>` for char output—this is the type of `cout`, `cerr` and `clog`.
- `iostream` is a `basic_iostream<char>` for char input and output.

We used the aliases `istream` and `ostream` in Chapter 11 when we overloaded the stream extraction and stream insertion operators. The library also defines versions of these for `wchar_t`-based I/O—named `wistream`, `wostream` and `wiostream`, respectively. We cover only the char-based streams here.

Standard Stream Objects `cin`, `cout`, `cerr` and `clog`

Predefined object `cin` is an `istream` that's connected to the **standard input device**—usually the keyboard. In a stream extraction operation (`>>`) like

```
int grade{0};  
std::cin >> grade; // data "flows" in the direction of the arrows
```

the compiler selects the appropriate overloaded stream extraction operator based on the variable `grade`'s type—`int` in this case. The `>>` operator is overloaded to input fundamental-type values, strings and pointer values.

The predefined object `cout` is an `ostream` that's connected to the **standard output device**. Standard output typically appears in

- a Command Prompt or PowerShell window in Microsoft Windows,
- a Terminal in macOS or Linux, or
- a shell window in Linux.

The stream insertion operator (`<<`) outputs its right operand to the standard output device:

```
std::cout << grade; // data "flows" in the direction of the arrows
```

The compiler selects the appropriate stream insertion operator for `grade`'s type—`<<` is overloaded to output data items of fundamental types, strings and pointer values.

The predefined object `cerr` is an `ostream` that's connected to the **standard error device**—typically the same device as the standard output device. Outputs to object `cerr` are **unbuffered**, meaning that each stream insertion to `cerr` performs its output immediately—this is appropriate for notifying a user promptly about errors.

The predefined object `clog` is an `ostream` that's connected to the **standard error device**. Outputs to `clog` are **buffered**. Each output might be held in a buffer (that is, an area in memory) until the buffer is filled or until the buffer is flushed. Buffering is an I/O



performance-enhancement technique.

19.3 Stream Output

`ostream` provides both formatted and unformatted output capabilities, including

- outputting standard data types with the stream insertion operator (`<<`);

- outputting characters via the `put` member function;
- **unformatted output** via the `write` member function;
- outputting integers in decimal, octal and hexadecimal formats;
- outputting floating-point values with various precisions, with **forced decimal points**, in **scientific notation** (e.g., `1.234567e-03`) and in **fixed notation** (e.g., `0.00123457`);
- outputting data **aligned** in fields of designated widths;
- outputting data in fields **padded** with specified characters; and
- outputting uppercase letters in scientific notation and **hexadecimal (base-16) notation**.

We'll demonstrate all of these capabilities in this chapter.

19.3.1 Output of `char*` Variables

Generally, you should avoid using pointers in favor of the modern C++ techniques we've discussed. In programs that require pointers, occasionally, you might want to print the addresses they contain (e.g., for debugging). The `<<` operator outputs a `char*` as a **null-terminated C-style string**. To output the **address**, cast the `char*` to a `void*` (Fig. 19.1, line 12). Operator `<<`'s `void*` version displays the pointer in an implementation-dependent manner—often as a hexadecimal number.¹ Figure 19.1 prints a `char*` variable as a null-terminated C-style string and an address. The output will vary by compiler and operating system. We say more about controlling the bases of numbers in Section 19.6.1 and Section 19.6.8.

```
1 // fig19_01.cpp
2 // Printing the address stored in a char* variable.
3 #include <iostream>
4
5 int main() {
6     const char* const word{"again"};
7
8     // display the value of char* variable word, then display
9     // the value of word after a static_cast to void*
10    std::cout << "Value of word is: " << word
11    << "\nValue of static_cast<const void*>(word) is: "
12    << static_cast<const void*>(word) << '\n';
13 }
```

```
Value of word is: again
Value of static_cast<const void*>(word) is: 00007FF611416410
```

Fig. 19.1 | Printing the address stored in a `char*` variable.

1. See the Number Systems appendix at <https://deitel.com/cpphtp11> to learn more about hexadecimal numbers.

19.3.2 Character Output Using Member Function `put`

The `basic_ostream` member function `put` outputs one character at a time. For example, the statement

```
std::cout.put('A');
```

displays a single character A. Calls to `put` can be chained, as in

```
std::cout.put('A').put('\n');
```

which outputs the letter A followed by a newline character. As with `<<`, the preceding statement executes in this manner because the dot operator (`.`) groups left-to-right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call. You also can call `put` with a numeric expression representing a character value. For example, the following statement outputs uppercase A:

```
std::cout.put(65);
```

19.4 Stream Input

`istream` provides formatted and unformatted input capabilities. The stream extraction operator (`>>`) normally skips white-space characters (such as blanks, tabs and newlines) in the input stream. Later, we'll see how to change this behavior.

Using the Result of a Stream Extraction as a Condition

After each input, the stream extraction operator returns a reference to the stream object that received the extraction message (e.g., `cin` in the expression `cin >> grade`). If that reference is used as a condition (e.g., in a `while` statement's loop-continuation condition), the stream's overloaded `bool` cast operator function is implicitly invoked to convert the reference into `true` or `false` value, based on the success or failure, respectively, of the last input operation. When an attempt is made to read past the end of a stream, the stream's overloaded `bool` cast operator returns `false` to indicate end-of-file. We used this capability in line 24 of Fig. 4.6.

19.4.1 `get` and `getline` Member Functions

The `get` member function with no arguments inputs and returns one character from the designated stream—including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file. This version of `get` returns EOF when end-of-file is encountered on the stream. EOF normally has the value `-1` and is defined in a header that's included in your code via stream library headers like `<iostream>`.

Using Member Functions `eof`, `get` and `put`

Figure 19.2 demonstrates member functions `eof` and `get` on input stream `cin` and member function `put` on output stream `cout`. This program uses `get` to read characters into the `int` variable `character`, so we can test for EOF. Function `get` returns an `int` because `char` can represent only nonnegative values on many platforms, and EOF is typically defined as `-1`. Line 10 prints the value of `cin.eof()`—initially `false`—before any inputs to show that end-of-file has not yet occurred on `cin`. You enter a line of text and press *Enter*, followed by the end-of-file indicator:

- $<Ctrl>z$ on Microsoft Windows systems or
- $<Ctrl>d$ on Linux and Mac systems.

Line 14 reads each character, which line 15 outputs to `cout` using member function `put`. When end-of-file is encountered, the `while` statement ends, and lines 19–20 display the integer value of the last character read (-1 for end-of-file) and the current value of `cin.eof()`, which now returns `true`, to show that end-of-file has been set on `cin`. Function `eof` returns `true` only after the program attempts to read past the last character in the stream.

```

1 // fig19_02.cpp
2 // get, put and eof member functions.
3 #include <iostream>
4 #include <iomanip>
5
6 int main() {
7     int character{0}; // use int, because char cannot represent EOF
8
9     // prompt user to enter line of text
10    std::cout << std::format("Before input, cin.eof(): {}", std::cin.eof())
11        << "\nEnter a sentence followed by Enter and end-of-file:\n";
12
13    // use get to read each character; use put to display it
14    while ((character = std::cin.get()) != EOF) {
15        std::cout.put(character);
16    }
17
18    // display end-of-file character
19    std::cout << std::format("\nEOF on this system is: {}\n", character)
20        << std::format("After EOF input, cin.eof(): {}\n", std::cin.eof());
21 }
```

```

Before input, cin.eof(): false
Enter a sentence followed by Enter and end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF on this system is: -1
After EOF input, cin.eof(): true

```

Fig. 19.2 | `get`, `put` and `eof` member functions.

Other `get` Versions

The `get` member function with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object on which the function is invoked.

A third version of `get` takes three arguments—a built-in array of `char`s, a size limit and a delimiter (with default value '`\n`'). This version can read multiple characters from the input stream. It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array argument. The delimiter is not placed in the character array. Rather, it remains in the input stream and will be the next

character read if the program performs more input. Thus, the result of a second consecutive get is an empty line (possibly a logic error) unless the delimiter character is removed from the input stream—which you can do simply by calling `cin.ignore()`.

Comparing `cin` and `cin.get`

Figure 19.3 compares input using the stream extraction operator with `cin` (line 14), which reads characters until a white-space character is encountered, and input using the three-argument version of `cin.get` with its third argument defaulted to the '\n' character.

```

1 // fig19_03.cpp
2 // Contrasting input of a string via cin and cin.get.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     // create two char arrays, each with 80 elements
8     constexpr int size{80};
9     char buffer1[size]{};
10    char buffer2[size]{};
11
12    // use cin to input characters into buffer1
13    std::cout << "Enter a sentence:\n";
14    std::cin >> buffer1;
15
16    // display buffer1 contents
17    std::cout << std::format("\nThe cin input was:\n{}\\n\\n", buffer1);
18
19    // use cin.get to input characters into buffer2
20    std::cin.get(buffer2, size);
21
22    // display buffer2 contents
23    std::cout << std::format("The cin.get input was:\n{}\\n", buffer2);
24 }
```

```

Enter a sentence:
Contrasting string input with cin and cin.get

The cin input was:
Contrasting

The cin.get input was:
string input with cin and cin.get
```

Fig. 19.3 | Contrasting input of a string via `cin` and `cin.get`.



Before C++20, line 14 could write past the end of `buffer1`—a potentially fatal logic error. In C++20, the `char array` overload of `operator>>` is a **function template** that the compiler instantiates using its `char array` argument's size. In line 14, the compiler knows that `buffer1` contains 80 characters (as defined in line 9), so it instantiates an `operator>>` function that limits the number of characters input to a maximum of 79, saving one array element for the C-style string's terminating '\0' character.

Using Member Function `getline`

Member function `getline` operates similarly to the third version of the `get` member function and inserts a null character after the line in the built-in array of chars. The `getline` function removes the delimiter from the stream (i.e., reads the character and discards it) but **does not store it in the character array**. The program of Fig. 19.4 uses `getline` to input a line of text (line 12). Again, you should avoid using built-in arrays and pointers. So, C++ also provides `std::getline` in the `<string>` header.² This version of `getline` reads data and places it into a `string` object.

```

1 // fig19_04.cpp
2 // Inputting characters using cin member function getline.
3 #include <iostream>
4 #include <iomanip>
5
6 int main() {
7     const int size{80};
8     char buffer[size]{ }; // create array of 80 characters
9
10    // input characters in buffer via cin function getline
11    std::cout << "Enter a sentence:\n";
12    std::cin.getline(buffer, size);
13
14    // display buffer contents
15    std::cout << std::format("\nYou entered:{}\n", buffer);
16 }
```

```

Enter a sentence:
Using the getline member function

You entered:
Using the getline member function
```

Fig. 19.4 | Inputting characters using `cin` member function `getline`.

19.4.2 `istream` Member Functions `peek`, `putback` and `ignore`

`istream` member function `ignore` reads and discards characters. It receives as arguments:

- a designated number of characters—the default argument value is 1—and
- a delimiter at which to stop ignoring characters—the default delimiter is `EOF`.

The function discards the specified number of characters. It discards fewer characters if the delimiter is encountered in the input stream.

The `putback` member function places the previous character obtained by a `get` from an input stream back into that stream. This is helpful in applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream for the next input operation.

The `peek` member function returns the next character from an input stream but does not remove the character from the stream.

2. “`std::getline`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/string/basic_string/getline.

19.5 Unformatted I/O Using `read`, `write` and `gcount`

Unformatted input/output is performed using `istream`'s `read` and `ostream`'s `write` member functions, respectively:

- `read` inputs bytes to a built-in array of chars in memory.
- `write` outputs bytes from a built-in array of chars.

These bytes are input or output simply as “raw” bytes—they are not formatted in any way. For example, the following call outputs the first 10 bytes of buffer, including null characters, if any, that would cause output with `cout` and `<<` to terminate:

```
char buffer[]{"HAPPY BIRTHDAY"};
std::cout.write(buffer, 10);
```

Similarly, the following call displays the first 10 characters of the alphabet:

```
std::cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

The `read` member function inputs a designated number of characters into a built-in array of chars. If fewer than the designated number of characters are read, `failbit` is set. Section 19.7 shows how to determine whether `failbit` has been set. Member function `gcount` reports the number of characters read by the last input operation.

Figure 19.5 demonstrates `istream` member functions `read` and `gcount` and `ostream` member function `write`. The program inputs 20 characters (from a longer input sequence) into the array `buffer` with `read` (line 10), determines the number of characters input with `gcount` (line 14) and outputs the characters in `buffer` with `write` (line 14).

```

1 // fig19_05.cpp
2 // Unformatted I/O using read, gcount and write.
3 #include <iostream>
4
5 int main() {
6     char buffer[80]{};
7     // create array of 80 characters
8     // use function read to input characters into buffer
9     std::cout << "Enter a sentence:\n";
10    std::cin.read(buffer, 20);
11
12    // use functions write and gcount to display buffer characters
13    std::cout << "\nThe sentence entered was:\n";
14    std::cout.write(buffer, std::cin.gcount());
15    std::cout << '\n';
16}
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 19.5 | Unformatted I/O using `read`, `gcount` and `write`.

19.6 Stream Manipulators

C++ provides various **stream manipulators** to specify formatting in streams. The stream manipulators provide capabilities such as

- setting the base for integer values
- setting field widths
- setting precision
- setting and unsetting format state
- setting the fill character in fields
- flushing streams
- inserting a newline into the output stream (and flushing the stream)
- inserting a null character into the output stream
- skipping white space in the input stream

The following table lists various stream manipulators that control a given stream's format state. We show examples of many of these stream manipulators in the next several sections.

Manipulator	Description
<code>skipws</code>	Skips white-space characters on an input stream. You can reset this setting with stream manipulator <code>noskipws</code> .
<code>left</code>	Left aligns output in a field. Padding characters appear to the right if necessary.
<code>right</code>	Right aligns output in a field. Padding characters appear to the left if necessary.
<code>internal</code>	In a field, this left aligns a number's sign and right aligns a number's value. Padding characters appear between the sign and the number if necessary.
<code>boolalpha</code>	Displays <code>bool</code> values as the word <code>true</code> or <code>false</code> . Similarly, <code>noboolalpha</code> sets the stream back to displaying <code>bool</code> values as 1 (true) and 0 (false).
<code>dec</code>	Treats integers as <code>decimal</code> (base 10) values.
<code>oct</code>	Treats integers as <code>octal</code> (base 8) values.
<code>hex</code>	Treats integers as <code>hexadecimal</code> (base 16) values.
<code>showbase</code>	Outputs a number's base before the number—0 for octals and 0x or 0X for hexadecimals. You can reset this with stream manipulator <code>noshowbase</code> .
<code>showpoint</code>	Forces floating-point numbers to display a decimal point. Normally, this is used with <code>fixed</code> to guarantee a certain number of digits to the right of the decimal point. You can reset this setting with stream manipulator <code>noshowpoint</code> .
<code>uppercase</code>	Displays hexadecimal integers with uppercase letters (i.e., X and A through F) and displays floating-point values in scientific notation with an uppercase E. You can reset this setting with stream manipulator <code>nouppercase</code> .
<code>showpos</code>	Precedes positive numbers by a plus sign (+). You can reset this with <code>noshowpos</code> .
<code>scientific</code>	Outputs floating-point values in <code>scientific notation</code> .
<code>fixed</code>	Outputs floating-point values in <code>fixed-point notation</code> with a specific number of digits to the right of the decimal point.

19.6.1 Integral Stream Base: dec, oct, hex and setbase

Integers typically are processed as decimal (base 10) values. To change this, you can insert the **hex** stream manipulator to set the base to hexadecimal (base 16) or insert the **oct** manipulator to set the base to octal (base 8). Insert the **dec** manipulator to reset the stream base to decimal. These **stream manipulators** are all **sticky**—that is, the settings remain in effect until you change them.

You also can set a stream's integer base via the **setbase parameterized stream manipulator** (header `<iomanip>`). A parameterized stream manipulator takes an argument—in this case, the value 10, 8, or 16 to set the base to decimal, octal or hexadecimal.³ The stream base value remains the same until changed explicitly, so **setbase** settings are sticky. Figure 19.6 demonstrates stream manipulators **hex** (line 14), **dec** (line 17), **oct** (line 18) and **setbase** (line 21).

```

1 // fig19_06.cpp
2 // Using stream manipulators dec, oct, hex and setbase.
3 #include <iomanip>
4 #include <iostream>
5
6 int main() {
7     int number{0};
8
9     std::cout << "Enter a decimal number: ";
10    std::cin >> number; // input number
11
12    // use hex stream manipulator to show hexadecimal number
13    std::cout << number << " in hexadecimal is: "
14    << std::hex << number << "\n";
15
16    // use oct stream manipulator to show octal number
17    std::cout << std::dec << number << " in octal is: "
18    << std::oct << number << "\n";
19
20    // use setbase stream manipulator to show decimal number
21    std::cout << std::setbase(10) << number << " in decimal is: "
22    << number << "\n";
23 }
```

```

Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20

```

Fig. 19.6 | Using stream manipulators **dec**, **oct**, **hex** and **setbase**.

19.6.2 Floating-Point Precision (**setprecision**, **precision**)

You can control the **precision** of floating-point numbers—that is, the number of digits to the right of the decimal point—with the **setprecision** stream manipulator or the **ost-**

3. The Number Systems appendix at <https://deitel.com/cphptp11> discusses the decimal, octal and hexadecimal number systems.

ream member function **precision**. Both are sticky—a call to either sets the precision for all subsequent output operations until the next precision-setting call. Calling member function **precision** with no argument returns the current precision setting. You can use this to save the current precision setting so you can restore it later. Figure 19.7 uses both member function **precision** (line 16) and the **setprecision** manipulator (line 24) to print a table that shows the square root of 2, with precision varying from 0 to 9. Stream manipulator **fixed** (line 12) forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function **precision** or stream manipulator **setprecision**.

```

1 // fig19_07.cpp
2 // Controlling precision of floating-point values.
3 #include <iomanip>
4 #include <iostream>
5 #include <cmath>
6
7 int main() {
8     double root2{std::sqrt(2.0)}; // calculate square root of 2
9
10    std::cout << "Square root of 2 with precisions 0-9.\n"
11        << "Precision set by ostream member function precision:\n";
12    std::cout << std::fixed; // use fixed-point notation
13
14    // display square root using ostream function precision
15    for (int places{0}; places <= 9; ++places) {
16        std::cout.precision(places);
17        std::cout << root2 << "\n";
18    }
19
20    std::cout << "\nPrecision set by stream manipulator setprecision:\n";
21
22    // set precision for each digit, then display square root
23    for (int places{0}; places <= 9; ++places) {
24        std::cout << std::setprecision(places) << root2 << "\n";
25    }
26 }
```

```

Square root of 2 with precisions 0-9.
Precision set by ostream member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

```

Fig. 19.7 | Controlling precision of floating-point values. (Part I of 2.)

```
Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 19.7 | Controlling precision of floating-point values. (Part 2 of 2.)

19.6.3 Field Width (`width`, `setw`)

The `width` member function (of classes `istream` and `ostream`) sets the **field width**—that is, the number of character positions in which a value should be output or the maximum number of characters that should be input. The function also returns the previous field width so you can save it and restore the value later. If the values output are narrower than the field width, **fill characters** are inserted as **padding**. When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs. The width setting is not sticky—it applies only for the next insertion or extraction. Afterward, the width is set implicitly to 0, so subsequent inputs or outputs will be performed with default settings. Calling `width` with no argument returns the current setting.

Figure 19.8 demonstrates the `width` member function for both input (lines 10 and 16) and output (line 14). For input into a `char` array, a maximum of **one fewer characters than the width are read**, saving one element for the null character to be placed at the end of the C-style string. Remember that stream extraction terminates when nonleading white space is encountered. When prompted for input in Fig. 19.8, enter a line of text and press *Enter* followed by end-of-file (*<Ctrl>z* on Microsoft Windows systems and *<Ctrl>d* on Linux and OS X systems). The **setw parameterized stream manipulator** also may be used to set the field width by inserting a call to it in a `cin` or `cout` statement.

```
1 // fig19_08.cpp
2 // width member function of classes istream and ostream.
3 #include <iostream>
4
5 int main() {
6     int widthValue{4};
7     char sentence[10]{};
8
9     std::cout << "Enter a sentence:\n";
10    std::cin.width(5); // input up to 4 characters from sentence
11}
```

Fig. 19.8 | `width` member function of class classes `istream` and `ostream`. (Part 1 of 2.)

```

12 // set field width, then display characters based on that width
13 while (std::cin >> sentence) {
14     std::cout.width(widthValue++);
15     std::cout << sentence << "\n";
16     std::cin.width(5); // input up to 4 more characters from sentence
17 }
18 }
```



```

Enter a sentence:
This is a test of the width member function
This
is
a
test
of
the
width
member
function
^Z
```

Fig. 19.8 | width member function of class classes `istream` and `ostream`. (Part 2 of 2.)

19.6.4 User-Defined Output Stream Manipulators

You can create your own stream manipulators. Figure 19.9 shows how to create and use custom nonparameterized output-stream manipulators `bell` (lines 7–9) and `tab` (lines 12–14). These are defined as functions with `ostream&` as the return type and parameter type. When lines 19 and 21 insert `tab` and `bell` in the output stream, their corresponding functions are called, which in turn output the `\a` (alert) and `\t` (tab) escape sequences, respectively. The `bell` manipulator does not display any text. Rather, it plays your system's alert sound.

```

1 // fig19_09.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream>
5
6 // bell manipulator (using escape sequence \a)
7 std::ostream& bell(std::ostream& output) {
8     return output << '\a'; // issue system beep
9 }
10
11 // tab manipulator (using escape sequence \t)
12 std::ostream& tab(std::ostream& output) {
13     return output << '\t'; // issue tab
14 }
15
```

Fig. 19.9 | Creating and testing user-defined, nonparameterized stream manipulators. (Part 1 of 2.)

```

16 int main() {
17     // use tab and bell manipulators
18     std::cout << "Testing the tab manipulator:\n"
19     << 'a' << tab << 'b' << tab << 'c' << '\n';
20
21     std::cout << "Testing the bell manipulator\n" << bell;
22 }
```

```

Testing the tab manipulator:
a      b      c
Testing the bell manipulator
```

Fig. 19.9 | Creating and testing user-defined, nonparameterized stream manipulators. (Part 2 of 2.)

19.6.5 Trailing Zeros and Decimal Points (showpoint)

Stream manipulator `showpoint` (Fig. 19.10) is a sticky setting that forces a floating-point number to display a decimal point and trailing zeros. For example, the floating-point value 79.0 prints as 79 without using `showpoint` and prints as 79.00000 using `showpoint`. The number of trailing zeros is determined by the current `precision`. To reset the `showpoint` setting, output the stream manipulator `noshowpoint`. The `default precision` of floating-point numbers is 6, which you can see in the output produced by lines 13–17. When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after the decimal point.

```

1 // fig19_10.cpp
2 // Displaying trailing zeros and decimal points in floating-point values.
3 #include <iostream>
4
5 int main() {
6     // display double values with default stream format
7     std::cout << "Before using showpoint"
8     << "\n9.9900 prints as: " << 9.9900
9     << "\n9.9000 prints as: " << 9.9000
10    << "\n9.0000 prints as: " << 9.0000;
11
12    // display double value after showpoint
13    std::cout << std::showpoint
14    << "\n\nAfter using showpoint"
15    << "\n9.9900 prints as: " << 9.9900
16    << "\n9.9000 prints as: " << 9.9000
17    << "\n9.0000 prints as: " << 9.0000 << '\n';
18 }
```

```

Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9
```

Fig. 19.10 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 1 of 2.)

```
After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 19.10 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 2 of 2.)

19.6.6 Alignment (`left`, `right` and `internal`)

Stream manipulators `left` and `right` enable fields to be **left-aligned** with padding characters to the right or **right-aligned** with padding characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 19.6.7). Figure 19.11 uses the `setw`, `left` and `right` manipulators to left-align and right-align integer data in a field—we wrap each field in quotes so you can see the leading and trailing space in the field.

```

1 // fig19_11.cpp
2 // Left and right alignment with stream manipulators left and right.
3 #include <iomanip>
4 #include <iostream>
5
6 int main() {
7     int x{12345};
8
9     // display x right aligned (default)
10    std::cout << "Default is right aligned:\n\""
11    << std::setw(10) << x << "\"";
12
13    // use left manipulator to display x left aligned
14    std::cout << "\n\nUse left to left align x:\n\""
15    << std::left << std::setw(10) << x << "\"";
16
17    // use right manipulator to display x right aligned
18    std::cout << "\n\nUse right to right align x:\n\""
19    << std::right << std::setw(10) << x << "\"\n";
20 }
```

```
Default is right aligned:
"      12345"

Use left to left align x:
"12345      "

Use right to right align x:
"      12345"
```

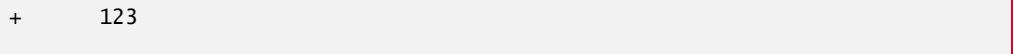
Fig. 19.11 | Left and right alignment with stream manipulators `left` and `right`.

Stream manipulator `internal` (Fig. 19.12; line 8) indicates that a number's **sign** should be **left-aligned** within a field, the number's magnitude should be **right-aligned** and

intervening spaces should be padded with the **fill character**. When using stream manipulator **showbase**, the **base** is left aligned. The **showpos** manipulator (line 8) forces the plus sign to print. To reset the **showpos** setting, output the stream manipulator **noshowpos**.

```

1 // fig19_12.cpp
2 // Printing an integer with internal spacing and plus sign.
3 #include <iomanip>
4 #include <iostream>
5
6 int main() {
7     // display value with internal spacing and plus sign
8     std::cout << std::internal << std::showpos
9     << std::setw(10) << 123 << "\n";
10 }
```



```
+      123
```

Fig. 19.12 | Printing an integer with internal spacing and plus sign.

19.6.7 Padding (**fill**, **setfill**)

The **fill** member function specifies the **fill character** to use with aligned fields and returns the previous fill character. Spaces are used for padding by default. The **setfill** manipulator also sets the **padding character**. Figure 19.13 demonstrates **fill** (line 29) and **setfill** (lines 33 and 38) to set the fill character.

```

1 // fig19_13.cpp
2 // Using member function fill and stream manipulator setfill to change
3 // the padding character for fields larger than the printed value.
4 #include <iomanip>
5 #include <iostream>
6
7 int main() {
8     int x{10000};
9
10    // display x
11    std::cout << x << " printed as int right and left aligned\n"
12    << "and as hex with internal alignment.\n"
13    << "Using the default pad character (space):\n";
14
15    // display x
16    std::cout << std::setw(10) << x << "\n";
17
18    // display x with left alignment
19    std::cout << std::left << std::setw(10) << x << "\n";
20 }
```

Fig. 19.13 | Using member function **fill** and stream manipulator **setfill** to change the padding character for fields larger than the printed values. (Part 1 of 2.)

```

21 // display x with base as hex with internal alignment
22 std::cout << std::showbase << std::internal << std::setw(10)
23     << std::hex << x << "\n\n";
24
25 std::cout << "Using various padding characters:\n";
26
27 // display x using padded characters (right alignment)
28 std::cout << std::right;
29 std::cout.fill('*');
30 std::cout << std::setw(10) << std::dec << x << "\n";
31
32 // display x using padded characters (left alignment)
33 std::cout << std::left << std::setw(10) << std::setfill('%')
34     << x << "\n";
35
36 // display x using padded characters (internal alignment)
37 std::cout << std::internal << std::setw(10)
38     << std::setfill('A') << std::hex << x << "\n";
39 }

```

```

10000 printed as int right and left aligned
and as hex with internal alignment.
Using the default pad character (space):
    10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710

```

Fig. 19.13 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the printed values. (Part 2 of 2.)

19.6.8 Integral Stream Base (dec, oct, hex, showbase)

C++ provides stream manipulators `dec`, `oct` and `hex` to specify that integers should display as decimal, hexadecimal and octal values, respectively. Integers display in decimal (base 10) by default. With stream extraction, integers prefixed with 0 are treated as octal values, integers prefixed with 0x or 0X are treated as hexadecimal values, and all other integers are treated as decimal values. Once you specify a stream's base, it processes all integers using that base until you specify a different one or until the program terminates.

Stream manipulator `showbase` causes octal numbers to be output with a leading 0 and hexadecimal numbers with either a leading 0x or a leading 0X—Section 19.6.10 shows that stream manipulator `uppercase` determines which option is chosen for hexadecimal values. Figure 19.14 demonstrates `showbase`. To reset the `showbase` setting, insert the stream manipulator `noshowbase` in the stream.

```

1 // fig19_14.cpp
2 // Stream manipulator showbase.
3 #include <iostream>
4
5 int main() {
6     int x{100};
7
8     // use showbase to show number base
9     std::cout << "Printing octal and hexadecimal values with showbase:\n"
10    << std::showbase;
11
12    std::cout << x << "\n"; // print decimal value
13    std::cout << std::oct << x << "\n"; // print octal value
14    std::cout << std::hex << x << "\n"; // print hexadecimal value
15 }
```

```

Printing octal and hexadecimal values with showbase:
100
0144
0x64

```

Fig. 19.14 | Stream manipulator `showbase`.

19.6.9 Floating-Point Numbers; Scientific and Fixed Notation (scientific, fixed)

When you display a floating-point number without specifying its format, its value determines the output format. Some numbers display in scientific notation and others in fixed-point notation. The sticky stream manipulators `scientific` and `fixed` control the output format of floating-point numbers:

- `scientific` forces a floating-point number to display in scientific format.
- `fixed` forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function `precision` or stream manipulator `setprecision`.

Figure 19.15 displays floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 15) and `fixed` (line 19). The exponent format in scientific notation might vary among compilers.

```

1 // fig19_15.cpp
2 // Floating-point values displayed in system default,
3 // scientific and fixed formats.
4 #include <iostream>
5
6 int main() {
7     double x{0.001234567};
8     double y{1.946e9};
9 }
```

Fig. 19.15 | Floating-point values displayed in system default, scientific and fixed formats. (Part 1 of 2.)

```

10 // display x and y in default format
11 std::cout << "Displayed in default format:\n" << x << '\t' << y;
12
13 // display x and y in scientific format
14 std::cout << "\n\nDisplayed in scientific format:\n"
15     << std::scientific << x << '\t' << y;
16
17 // display x and y in fixed format
18 std::cout << "\n\nDisplayed in fixed format:\n"
19     << std::fixed << x << '\t' << y << "\n";
20 }

```

```

Displayed in default format:
0.00123457    1.946e+09

Displayed in scientific format:
1.234567e-03    1.946000e+09

Displayed in fixed format:
0.001235    1946000000.00000

```

Fig. 19.15 | Floating-point values displayed in system default, scientific and fixed formats. (Part 2 of 2.)

19.6.10 Uppercase/Lowercase Control (`uppercase`)

Stream manipulator `uppercase` outputs an uppercase X with hexadecimal-integer values or an uppercase E with scientific-notation floating-point values (Fig. 19.16; line 11). Using `uppercase` also displays the hexadecimal digits A–F in uppercase. These appear in lowercase by default. To reset the `uppercase` setting, output `nouppercase`.

```

1 // fig19_16.cpp
2 // Stream manipulator uppercase.
3 #include <iostream>
4
5 int main() {
6     std::cout << "Printing uppercase letters in scientific\n"
7         << "notation exponents and hexadecimal values:\n";
8
9     // use std::uppercase to display uppercase letters; use std::hex and
10    // std::showbase to display hexadecimal value and its base
11    std::cout << std::uppercase << 4.345e10 << "\n"
12        << std::hex << std::showbase << 123456789 << "\n";
13 }

```

```

Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
0X75BCD15

```

Fig. 19.16 | Stream manipulator `uppercase`.

19.6.11 Specifying Boolean Format (`boolalpha`)

C++ `bool` values may be `false` or `true`. Recall that 0 also indicates `false`, and any nonzero value indicates `true`. A `bool` value displays as 0 or 1 by default. You can use stream manipulator `boolalpha` to set the output stream to display `bool` values as the strings "true" and "false" and stream manipulator `noboolalpha` to set the output stream back to displaying `bool` values as the integers 0 and 1. Figure 19.17 demonstrates these stream manipulators. Line 9 displays `booleanValue` (which line 6 sets to `true`) as an integer. Line 13 uses `boolalpha` to display the `bool` value as a string. Lines 16–17 then change the `booleanValue` to `false` and use manipulator `noboolalpha`, so line 20 can display the `bool` value as an integer. Line 24 uses manipulator `boolalpha` to display the `bool` value as a string. Both `boolalpha` and `noboolalpha` are sticky settings.

```

1 // fig19_17.cpp
2 // Stream manipulators boolalpha and noboolalpha.
3 #include <iostream>
4
5 int main() {
6     bool booleanValue{true};
7
8     // display default true booleanValue
9     std::cout << "booleanValue is " << booleanValue;
10
11    // display booleanValue after using boolalpha
12    std::cout << "\nbooleanValue (after using boolalpha) is "
13        << std::boolalpha << booleanValue;
14
15    std::cout << "\n\nswitch booleanValue and use noboolalpha\n";
16    booleanValue = false; // change booleanValue
17    std::cout << std::noboolalpha; // use noboolalpha
18
19    // display default false booleanValue after using noboolalpha
20    std::cout << "\nbooleanValue is " << booleanValue;
21
22    // display booleanValue after using boolalpha again
23    std::cout << "\nbooleanValue (after using boolalpha) is "
24        << std::boolalpha << booleanValue << "\n";
25 }
```

```

booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false

```

Fig. 19.17 | Stream manipulators `boolalpha` and `noboolalpha`.

19.6.12 Setting and Resetting the Format State via Member Function flags

You've seen how to use stream manipulators to change output format characteristics, but how do you return an output stream's format to its previous state after changing its format? Member function `flags` without an argument returns the current **format state** settings as an `fmt::flags` object. Member function `flags` with an `fmt::flags` argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that `flags` returns might vary among compilers. Figure 19.18 uses member function `flags` to save the stream's original format state (line 16), then restore the original format settings (line 24). The capabilities for capturing the format state and restoring it are not required in C++20 text formatting. Each `std::format` call's formatting is used only in that call and does not affect any other `std::format` call, thus simplifying output formatting.

```

1 // fig19_18.cpp
2 // flags member function.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     int integerValue{1000};
8     double doubleValue{0.0947628};
9
10    // display flags value, int and double values (original format)
11    std::cout << std::format("flags value: {}\n", std::cout.flags())
12        << "int and double in original format:\n"
13        << integerValue << '\t' << doubleValue << "\n\n";
14
15    // save original format, then change the format
16    auto originalFormat{std::cout.flags()};
17    std::cout << std::showbase << std::oct << std::scientific;
18
19    // display flags value, int and double values (new format)
20    std::cout << std::format("flags value: {}\n", std::cout.flags())
21        << "int and double in new format:\n"
22        << integerValue << '\t' << doubleValue << "\n\n";
23
24    std::cout.flags(originalFormat); // restore format
25
26    // display flags value, int and double values (original format)
27    std::cout << std::format("flags value: {}\n", std::cout.flags())
28        << "int and double in original format:\n"
29        << integerValue << '\t' << doubleValue << "\n";
30 }
```

Fig. 19.18 | `flags` member function. (Part I of 2.)

```

flags value: 513
int and double in original format:
1000    0.0947628

flags value: 5129
int and double in new format:
01750   9.476280e-02

flags value: 513
int and double in original format:
1000    0.0947628

```

Fig. 19.18 | flags member function. (Part 2 of 2.)

19.7 Stream Error States

Each stream object contains a set of **state bits** representing the stream's state—sticky format settings, error indicators, etc. You can use this information to test, for example, whether an input was successful. These state bits are defined in class `ios_base`—the base class of the stream classes. Stream extraction sets the stream's **failbit** to true if the wrong type of data is input. Similarly, stream extraction sets the stream's **badbit** to true if the operation fails in an unrecoverable manner—for example, if a disk fails when a program is reading a file from that disk. Figure 19.19 shows how to use bits like `failbit` and `badbit` to determine a stream's state.⁴ See the Bit Manipulation appendix at <https://deitel.com/cpphtp11> for more information on bits and bit manipulation.

```

1 // fig19_19.cpp
2 // Testing error states.
3 #include <iostream>
4
5 int main() {
6     int integerValue{0};
7
8     // display results of cin functions
9     std::cout << std::boolalpha << "Before a bad input operation:"
10    << "\n    cin.rdstate(): " << std::cin.rdstate()
11    << "\n    cin.eof(): " << std::cin.eof()
12    << "\n    cin.fail(): " << std::cin.fail()
13    << "\n    cin.bad(): " << std::cin.bad()
14    << "\n    cin.good(): " << std::cin.good()
15    << "\n\nExpects an integer, but enter a character: ";
16
17    std::cin >> integerValue; // enter character value
18

```

Fig. 19.19 | Testing error states. (Part 1 of 2.)

4. The actual values output by this program may vary among compilers.

```
19 // display results of cin functions after bad input
20 std::cout << "\nAfter a bad input operation:"
21     << "\ncin.rdstate(): " << std::cin.rdstate()
22     << "\n    cin.eof(): " << std::cin.eof()
23     << "\n    cin.fail(): " << std::cin.fail()
24     << "\n    cin.bad(): " << std::cin.bad()
25     << "\n    cin.good(): " << std::cin.good();
26
27 std::cin.clear(); // clear stream
28
29 // display results of cin functions after clearing cin
30 std::cout << "\n\nAfter cin.clear()"
31     << "\ncin.fail(): " << std::cin.fail()
32     << "\ncin.good(): " << std::cin.good() << "\n";
33 }
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): false
    cin.fail(): false
    cin.bad(): false
    cin.good(): true

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): false
    cin.fail(): true
    cin.bad(): false
    cin.good(): false

After cin.clear()
cin.fail(): false
cin.good(): true
```

Fig. 19.19 | Testing error states. (Part 2 of 2.)

Member Function eof

The program begins by displaying the stream's state before receiving any input from the user (lines 9–14). Line 11 uses member function `eof` to determine whether the end-of-file has been encountered on the stream. In this case, the function returns 0 (`false`). The function checks the value of the stream's `eofbit` data member, which is set to `true` for an input stream after the end-of-file is encountered after an attempt to extract data beyond the end of the stream.

Member Function fail

Line 12 uses the `fail` member function to determine whether a stream operation has failed. The function checks the value of the stream's `failbit` data member, which is set to `true` on a stream when a format error occurs and, as a result, no characters are input. For example, this might occur when you attempt to read a number, but the user enters a

string. In this case, the function returns 0 (`false`). When such an error occurs on input, the characters are not lost. Usually, recovering from such input errors is possible.

Member Function `bad`

Line 13 uses the `bad` member function to determine whether a stream operation failed. The function checks the value of the stream’s `badbit` data member, which is set to `true` for a stream when an error occurs that results in the loss of data—such as reading from a file when the disk on which the file is stored fails. In this case, the function returns 0 (`false`). Generally, such serious failures are nonrecoverable.

Member Function `good`

Line 14 uses the `good` member function, which returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. The function checks the stream’s `goodbit`, which is set to `true` for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set to `true` for the stream. In this case, the function returns 1 (`true`). I/O operations should be performed only on “good” streams.

Member Function `rdstate`

The `rdstate` member function (line 10) returns the stream’s overall error state as an integer value. The function’s return value could be tested, for example, by a `switch` statement that examines `eofbit`, `badbit`, `failbit` and `goodbit`. The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`—using these functions does not require you to be familiar with particular status bits.

Causing an Error in the Input Stream and Redisplaying the Stream’s State

Line 17 reads a value into an `int` variable. Enter a string rather than an `int` to force an error to occur in the input stream. At this point, the input fails, and lines 20–25 once again call the stream’s state functions. In this case, `fail` returns 1 (`true`) because the input failed. Function `rdstate` also returns a nonzero value (`true`) because at least one of the member functions `eof`, `bad` and `fail` returned `true`. Once an error occurs in the stream, function `good` returns 0 (`false`).

Clearing the Error State, So You May Continue Using the Stream

After an error occurs, you can no longer use the stream until you reset its error state. The `clear` member function (line 27) is used to restore a stream’s state to “good” so that I/O may proceed on that stream. Lines 30–32 show that `fail` returns 0 (`false`) and `good` returns 1 (`true`), so the input stream can be used again.

The default argument for `clear` is `goodbit`, so the statement

```
std::cin.clear();
```

clears `cin` and sets `goodbit` for the stream. The statement

```
std::cin.clear(ios::failbit)
```

sets the `failbit`. You might want to do this when performing input on `cin` with a user-defined type and encountering a problem. The name `clear` might seem inappropriate in this context, but it’s correct.

Overloaded Operators ! and bool

Overloaded operators can be used to test a stream's state in conditions. The operator `!` member function— inherited into the stream classes from class `basic_ios`— returns `true` if the `badbit`, the `failbit` or both are true. The operator `bool` member function returns `false` if the `badbit` is `true`, the `failbit` is `true` or both are `true`. These functions are useful in I/O processing when a `true/false` condition is being tested under the control of a selection statement or iteration statement. For example, you could use an `if` statement of the form

```
if (!std::cin) {  
    // process invalid input stream  
}
```

to execute code if `cin`'s stream is invalid due to a failed input. Similarly, you've already seen a `while` condition of the form

```
while (std::cin >> variableName) {  
    // process valid input  
}
```

which enables the loop to execute as long as each input operation is successful and terminates the loop if an input fails or the end-of-file indicator is encountered.

19.8 Tying an Output Stream to an Input Stream

Interactive command-line applications generally use an `istream` for input and an `ostream` for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only

- when the buffer fills
- when outputs are flushed explicitly by the program or
- automatically at the end of the program.

C++ provides member function `tie` to synchronize (i.e., “tie together”) an `istream` and an `ostream` to ensure that outputs happen (that is, they are flushed) before subsequent inputs. The call

```
std::cin.tie(&cout);
```

ties `cout` (an `ostream`) to `cin` (an `istream`). This particular call is redundant because C++ performs this operation automatically for the standard output and input streams. However, you might use this with other input/output stream pairs. To untie an input stream, `inputStream`, from an output stream, use the call

```
inputStream.tie(0);
```

19.9 C++20 Text Formatting

As we mentioned in Chapter 3, C++20 provides powerful string-formatting capabilities via the `format` function (in header `<format>`). These capabilities greatly simplify formatting by using a concise syntax that's based on the Python programming language's text for-

matting. As you've seen in this chapter, pre-C++20 output formatting is quite verbose. C++20 text-formatting capabilities are more concise and more powerful.

19.9.1 C++20 std::format Presentation Types

When you specify a placeholder for a value in a format string, the `std::format` function assumes the value should be displayed as a string unless you specify another type. In some cases, the type is required—for example, if you want to specify precision for a floating-point number or change the base in which to display an integer. In these cases, you can specify the **presentation type** in each placeholder. Figure 19.20 shows the various presentation types.

```

1 // fig19_20.cpp
2 // C++20 text formatting presentation types.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     // floating-point presentation types
8     std::cout << "Display 17.489 with default, .1 and .2 precisions:\n"
9     << std::format("f: {0:f}\n.1f: {0:.1f}\n.2f: {0:.2f}\n\n", 17.489);
10
11    std::cout << "Display 10000000000000000.0 with f, e, g and a:\n"
12    << std::format("f: {0:f}\ne: {0:e}\ng: {0:g}\na: {0:a}\n\n",
13        1000000000000000.0);
14
15    // integer presentation types; # displays a base prefix
16    std::cout << "Display 100 with d, #b, #o and #x:\n"
17    << std::format(
18        "d: {0:d}\n#b: {0:#b}\n#o: {0:#o}\n#x: {0:#x}\n\n", 100);
19
20    // character presentation type
21    std::cout << "Display 65 and 97 with c:\n"
22    << std::format("{:c} {:c}\n\n", 65, 97);
23
24    // string presentation type
25    std::cout << "Display \"hello\" with s:\n"
26    << std::format("{:s}\n", "hello");
27 }
```

```

Display 17.489 with default, .1 and .2 precisions:
f: 17.489000
.1f: 17.5
.2f: 17.49

Display 10000000000000000.0 with f, e, g and a
f: 1000000000000000.000000
e: 1.000000e+16
g: 1e+16
a: 1.1c37937e08p+53

```

Fig. 19.20 | C++20 text-formatting presentation types. (Part 1 of 2.)

```
Display 100 with d, #b, #o and #x:  
d: 100  
#b: 0b1100100  
#o: 0144  
#x: 0x64  
  
Display 65 and 97 with c:  
A a  
  
Display "hello" with s:  
hello
```

Fig. 19.20 | C++20 text-formatting presentation types. (Part 2 of 2.)

Floating-Point Values and C++20 Presentation Type f

You've used the **presentation type f** to format floating-point values. Formatting is **type-dependent**, so this presentation type is required to specify a floating-point number's precision. Line 9 uses the f presentation type to format the double value 17.489 in the default precision (6) and rounded to the tenths and hundredths positions. Function `std::format` uses presentation types to determine whether the other formatting options are allowed for a given type. For all the presentation types, their formatting options and the order in which the options must be specified, see

<https://en.cppreference.com/w/cpp/utility/format/formatter>

Indexing Arguments By Position

In line 9, note the 0 to the left of each format specifier's colon (:). You can reference the arguments after the format string **positionally** by their index numbers starting from index 0. This allows you to

- **reference the same argument multiple times**, as we did three times in line 9, or
- **reference the arguments in any order**, which can be helpful when localizing applications for spoken languages that order words differently in sentences.

Floating-Point Values and C++20 Presentation Types e, g and a

You also can format floating-point values using the following presentation types, as shown in lines 12–13:

- **e or E**—These use **exponential (scientific) notation** to format floating-point values. The exponent is preceded by an e or E in the formatted string. The value 1.000000e+16 in this program's output is equivalent to

1.000000 × 10¹⁶

- **g or G**—These choose between fixed-point notation and exponential notation based on the value's magnitude. For exponential notation, g displays a lowercase e, and G displays an uppercase E.
- **a or A**—These format floating-point values in hexadecimal notation with lowercase letters (a) or uppercase letters (A), respectively.

C++20 Integer Presentation Types

Lines 17–18 display the `int` value 100 using various integer number systems:⁵

- **d**—Displays an integer in decimal (base 10) format.
- **b or B**—These display an integer in binary (base 2) format with a lowercase **b** or uppercase **B** when a binary value is displayed with its base (Section 19.9.3).
- **o presentation type**—Displays an integer in octal (base 8) format.
- **x or X presentation type**—These display an integer in hexadecimal (base 16) format with a lowercase or uppercase letters, respectively.

C++20 Character Presentation Type

The **c presentation type** formats an integer character code as the corresponding character, as shown in line 22.

C++20 String Presentation Type

When a placeholder does not specify a presentation type, the default is to format the corresponding value as a string. The **s presentation type** indicates that the corresponding value must specifically be a string, an expression that produces a string or a string literal, as in line 26.

C++20 Locale-Specific Numeric and `bool` Formatting

If your application requires locale-specific formatting of numeric or `bool` values, precede the integer or floating-point presentation type with `L`.

19.9.2 C++20 `std::format` Field Widths and Alignment

Previously you used **field widths** to format text in a specified number of character positions. Figure 19.21 demonstrates field widths, default alignments and explicit alignments. We enclose each formatted value in brackets `[]` so you can better see the formatting results. By default, `std::format` **right-aligns numbers** and **left-aligns strings**, as demonstrated by line 8. For values with fewer characters than the field width, the remaining character positions are filled with spaces.⁶ Values with more characters than the specified field width use as many character positions as they need.

```

1 // fig19_21.cpp
2 // C++20 text formatting with field widths and alignment.
3 #include <format>
4 #include <iostream>
5
6 int main() {

```

Fig. 19.21 | C++20 text-formatting with field widths and alignment. (Part 1 of 2.)

-
5. See the Number Systems appendix at <https://deitel.com/cpphtp11> for information about the binary, octal and hexadecimal number systems.
 6. C++20 allows you to specify any fill character (other than `{` and `}`, which delimit placeholders) immediately to the right of the format specifier's colon `(:)`.

```

7   std::cout << "Default alignment with field width 10:\n"
8     << std::format("[{:10d}]\n[{:10f}]\n[{:10}]\n\n", 27, 3.5, "hello");
9
10  std::cout << "Specifying left or right alignment in a field:\n"
11    << std::format("[{:<15d}]\n[{:<15f}]\n[{:>15}]\n\n",
12      27, 3.5, "hello");
13
14  std::cout << "Centering text in a field:\n"
15    << std::format("[{:^7d}]\n[{:^7.1f}]\n[{:^7}]\n", 27, 3.5, "hello");
16 }

```

```

Default alignment with field width 10:
[ 27]
[ 3.500000]
[hello  ]

Specifying left or right alignment in a field:
[27          ]
[3.500000    ]
[        hello]

Centering text in a field:
[ 27  ]
[ 3.5  ]
[ hello ]

```

Fig. 19.21 | C++20 text-formatting with field widths and alignment. (Part 2 of 2.)

Recall that you can specify left-alignment and right-alignment with `<` and `>`. Lines 11–12 **left-align** the numeric values 27 and 3.5 and **right-align** the string "hello" in fields of 15 characters.

The I/O streams output formatting shown earlier in this chapter does not support **center-aligning** text, but `std::format` can do this conveniently with `^`, as shown in line 15. Centering attempts to spread the unoccupied character positions equally to the left and right of the formatted value. `std::format` places the extra space to the right if an odd number of character positions remain, as you can see for the value 27, which has two spaces to its left and three to its right.

19.9.3 C++20 `std::format` Numeric Formatting

Figure 19.22 demonstrates various C++20 numeric formatting capabilities. By default, negative numeric values display with a `-` sign. Sometimes it's desirable to force a `+` sign to display for a positive number. A `+` in the format specifier (line 8) indicates that the numeric value should always be preceded by a sign (`+` or `-`). To fill the field's remaining characters with `0`s rather than spaces, place `0` before the field width and *after* the `+` if there is one, as in line 8's second format specifier. A space in the format specifier (as in line 11's second and third format specifiers) indicates that positive numbers should show a space character in the sign position. This is useful for aligning positive and negative values for display purposes. Note that the two values with a space in their format specifiers align. If a field width is specified, place the space before the field width. To precede a binary, octal or hexadecimal number with its base, use `#` in the format specifier as in line 14.

```

1 // fig19_22.cpp
2 // C++20 text formatting numeric formatting options.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     std::cout << "Displaying signs and padding with Leading Os:\n"
8     << std::format("[{:+10d}]\n[{:+10d}]\n\n", 27);
9
10    std::cout << "Displaying a space before a positive value:\n"
11    << std::format("{0:d}\n{0: d}\n{1: d}\n\n", 27, -27);
12
13    std::cout << "Displaying a base indicator before a number:\n"
14    << std::format("{0:d}\n{0:#b}\n{0:#o}\n{0:#x}\n", 100);
15 }
```

```

Displaying signs and padding with leading Os:
[      +27]
[+000000027]

Displaying a space before a positive value:
27
 27
-27

Displaying a base indicator before a number:
100
0b1100100
0144
0x64

```

Fig. 19.22 | C++20 text-formatting numeric formatting options.

19.9.4 C++20 std::format Field Width and Precision Placeholders

You can programmatically specify field widths and precisions using **nested placeholders** in a format specifier. Figure 19.23 displays the `double` value `123.456` in a field of 8 characters with precisions of 0–4. In line 13, the argument `value` is the number to format. In the format specifier "`{:{}.{}f}`", the nested placeholders to the right of the colon (:) are replaced left-to-right by the values of the arguments `width` and `precision`, respectively.

```

1 // fig19_23.cpp
2 // C++20 text formatting field width and precision placeholders.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     std::cout << "Demonstrating field width and precision placeholders:\n";
8
9     double value{123.456};
10    int width{8};
```

Fig. 19.23 | C++20 text-formatting field width and precision placeholders.

```
11
12     for (int precision{0}; precision < 5; ++precision) {
13         std::cout << std::format("{:{}.{:}f}\n", value, width, precision);
14     }
15 }
```

Demonstrating field width and precision placeholders:

```
123
123.5
123.46
123.456
123.4560
```

Fig. 19.23 | C++20 text-formatting field width and precision placeholders.

19.10 Wrap-Up

This chapter showed C++ input/output formatting with streams. You learned about the stream-I/O classes and predefined objects. We discussed `ostream`'s formatted and unformatted output capabilities performed by the `put` and `write` functions. You learned about `istream`'s formatted and unformatted input capabilities performed by the `eof`, `get`, `getline`, `peek`, `putback`, `ignore` and `read` functions. We discussed stream manipulators and member functions that perform formatting tasks:

- `dec`, `oct`, `hex` and `setbase` for displaying integers
- `precision` and `setprecision` for controlling floating-point precision
- and `width` and `setw` for setting field width.

You also learned additional formatting with `iostream` manipulators and member functions:

- `showpoint` for displaying decimal point and trailing zeros
- `left`, `right` and `internal` for alignment
- `fill` and `setfill` for padding
- `scientific` and `fixed` for displaying floating-point numbers in scientific and fixed notation
- `uppercase` for uppercase/lowercase control
- `boolalpha` for specifying Boolean format
- and `flags` and `fmtflags` for resetting the format state.

You'll encounter many of the preceding capabilities in legacy C++ code.

Finally, we presented many of C++20's more concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.

Self-Review Exercises

- 19.1** Fill in the blanks in each of the following statements.
- C++ uses _____ I/O—each I/O operation is executed in a manner sensitive to the data type.
 - The `<iostream>` header defines the _____, _____, _____ and _____ objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively.
 - The `<>` operator outputs a `char*` as a null-terminated C-style string. To output the address, cast the `char*` to a _____.
 - The `oct`, `dec` and `hex` stream manipulators are all _____—the settings remain in effect until you change them.
 - The `left` and `right` stream manipulators enable fields to be left-aligned with padding characters to the right or right-aligned with padding characters to the left, respectively. The padding character is specified by the _____ member function or the _____ parameterized stream manipulator.
 - Stream manipulator `uppercase` outputs an uppercase X with hexadecimal-integer values or an uppercase E with scientific-notation floating-point values. Using `uppercase` also displays the hexadecimal digits A–F in uppercase. These appear in lowercase by default. To reset the `uppercase` setting, output _____.
 - After an error occurs, you can no longer use the stream until you reset its error state. The _____ member function is used to restore a stream's state to “good” so that I/O may proceed on that stream.
 - I/O streams output formatting does not support center-aligning text, but `std::format` can do this conveniently with _____.
 - You can programmatically specify field widths and precisions using _____ in a format specifier.
- 19.2** State whether each of the following is *true* or *false*. If the answer is *false*, explain why.
- C++ supports Unicode via the types `wchar_t`, `char16_t` and `char32_t` and `char8_t` (C++20).
 - The predefined object `clog` is an `ostream` that's connected to the standard error device. Outputs to object `clog` are unbuffered.
 - The `width` member function of an `ostream` sets the field width—that is, the number of character positions in which a value should be output. When a field is not sufficiently wide to handle outputs, the outputs are truncated.
 - C++ `bool` values may be `false` or `true`. Recall that 0 also indicates `false`, and any nonzero value indicates `true`. A `bool` value displays as 0 or 1 by default. You can use stream manipulator `boolalpha` to set the output stream to display `bool` values as the strings "true" and "false" and stream manipulator `noboolalpha` to set the output stream back to displaying `bool` values as the integers 0 and 1.
 - The `good` member function returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. The function checks the stream's `goodbit`, which is set to `true` for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set to `true`.

for the stream. In this case, the function returns 1 (true). I/O operations should be performed only on “good” streams.

- f) You can specify the precision with any `std::format` presentation type.
- g) The `s` presentation type indicates that `std::format`’s corresponding value must specifically be a string, an expression that produces a string or a string literal.

19.3 (Write a C++ Statement) For each of the following, write a single statement that performs the indicated task. Do not use `std::format`.

- a) Output the string "Enter your name: ".
- b) Use a stream manipulator that causes the exponent in scientific notation and the letters in hexadecimal values to print in capital letters.
- c) Output the address of the variable `myString` of type `char*`.
- d) Use a stream manipulator to ensure that floating-point values print in scientific notation.
- e) Output the address in variable `integerPtr` of type `int*`.
- f) Use a stream manipulator such that when integer values are output, the integer base for octal and hexadecimal values is displayed.
- g) Output the value pointed to by `floatPtr` of type `float*`.
- h) Use a stream member function to set the fill character to '*' for printing in field widths larger than the values being output. Repeat this statement with a stream manipulator.
- i) Output the characters 'O' and 'K' in one statement with `ostream` function `put`.
- j) Get the value of the next character to input without extracting it from the stream.
- k) Input a single character into variable `charValue` of type `char`, using the `istream` member function `get` in two different ways.
- l) Input and discard the next six characters in the input stream.
- m) Use `istream` member function `read` to input 50 characters into `char` array `line`.
- n) Read 10 characters into character array `name`. Stop reading characters if the '.' delimiter is encountered. Do not remove the delimiter from the input stream. Write another statement that performs this task and removes the delimiter from the input.
- o) Use the `istream` member function `gcount` to determine the number of characters input into character array `line` by the last call to `istream` member function `read`, and output that number of characters, using `ostream` member function `write`.
- p) Output 124, 18.376, 'Z', 1000000 and "String" separated by spaces.
- q) Display `cout`’s current precision setting.
- r) Input an integer value into `int` variable `months` and a floating-point value into `float` variable `percentageRate`.
- s) Print 1.92, 1.925 and 1.9258 separated by tabs and with 3 digits of precision, using a stream manipulator.
- t) Print integer 100 in octal, hexadecimal and decimal, using stream manipulators and separated by tabs.

- u) Print integer 100 in decimal, octal and hexadecimal separated by tabs, using a stream manipulator to change the base.
- v) Print 1234 right aligned in a 10-digit field.
- w) Read characters into character array line until the character 'z' is encountered, up to a limit of 20 characters (including a terminating null character). Do not extract the delimiter character from the stream.
- x) Use integer variables x and y to specify the field width and precision used to display the double value 87.4573, and display the value.

Answers to Self-Review Exercises

19.1 a) type-safe. b) `cin`, `cout`, `cerr`, `clog`. c) `void*`. d) sticky. e) `fill`, `setfill`. f) `nouppercase`. g) `clear`. h) `^`. i) nested placeholders.

19.2 a) True. b) False. Actually, `cerr`'s outputs are unbuffered—`clog`'s outputs are buffered. c) False. Actually, when a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs. d) True. e) True. f) False. Actually, precision may be specified only with the floating-point presentation types `f`, `e` or `E`. g) True.

- 19.3**
- a) `cout << "Enter your name: ";`
 - b) `cout << uppercase;`
 - c) `cout << static_cast<void*>(myString);`
 - d) `cout << scientific;`
 - e) `cout << integerPtr;`
 - f) `cout << showbase;`
 - g) `cout << *floatPtr;`
 - h) `cout.fill('*');`
`cout << setfill('*');`
 - i) `cout.put('0').put('K');`
 - j) `cin.peek();`
 - k) `charValue = cin.get();`
`cin.get(charValue);`
 - l) `cin.ignore(6);`
 - m) `cin.read(line, 50);`
 - n) `cin.get(name, 10, '.');`
`cin.getline(name, 10, '.');`
 - o) `cout.write(line, cin.gcount());`
 - p) `cout << 124 << ' ' << 18.376 << " Z " << 1000000 << " String";`
 - q) `cout << cout.precision();`
 - r) `cin >> months >> percentageRate;`
 - s) `cout << setprecision(3) << 1.92 << '\t' << 1.925 << '\t' << 1.9258;`
 - t) `cout << oct << 100 << '\t' << hex << 100 << '\t' << dec << 100;`
 - u) `cout << 100 << '\t' << setbase(8) << 100 << '\t' << setbase(16)`
`<< 100;`
 - v) `cout << setw(10) << 1234;`
 - w) `cin.get(line, 20, 'z');`
 - x) `cout << setw(x) << setprecision(y) << 87.4573;`

Exercises

19.4 Fill in the blanks in each of the following statements.

- a) C++ provides both “low-level” and “high-level” I/O capabilities. Low-level _____ I/O capabilities specify that some number of bytes should be transferred device-to-memory or memory-to-device. Higher-level _____ I/O groups bytes into meaningful units, such as integers, floating-point numbers, characters, strings and custom types.
- b) The predefined object `clog` is an `ostream` that’s connected to the standard error device. Outputs to `clog` are _____—they might be held temporarily in memory until that area is filled or flushed. This is an I/O performance-enhancement technique.
- c) An input stream’s _____ member function returns the next character in the stream but does not remove the character from the stream.
- d) You can set a stream’s integer base via the _____ parameterized stream manipulator (header `<iomanip>`).
- e) The _____ stream manipulator is a sticky setting that forces a floating-point number to display a decimal point and trailing zeros.
- f) The _____ stream manipulator indicates that a number’s sign should be left-aligned within a field, the number’s magnitude should be right-aligned and intervening spaces should be padded with the fill character.
- g) The _____ member function returns the stream’s overall error state as an integer value.
- h) The _____ presentation type formats an integer character code as the corresponding character.

19.5 State whether each of the following is *true* or *false*. If the answer is *false*, explain why.

- a) Member function `getline` inserts a null character after the line in the built-in array of `char`s.
- b) The precision of a floating-point number specifies the total number of digits in the number.
- c) Stream extraction sets the stream’s `badbit` to `true` if the wrong type of data is input.
- d) When you specify a placeholder for a value in a format string, the `std::format` function assumes the value should be displayed as a string unless you specify another type.
- e) The presentation types `f` and `F` use exponential (scientific) notation to format floating-point values.
- f) By default, `std::format` left-aligns numbers and right-aligns strings.

19.6 (*Write C++ Statements*) Write a statement for each of the following—do not use the `std::format` function:

- a) Print integer 40000 left aligned in a 15-digit field.
- b) Read a string into character array variable `state`.
- c) Print 200 with and without a sign.
- d) Print the decimal value 100 in hexadecimal form preceded by `0x`.

- e) Read characters into array charArray until the character 'p' is encountered, up to a limit of 10 characters (including the terminating null character). Extract the delimiter from the input stream, and discard it.
- f) Print 1.234 in a 9-digit field with preceding zeros.

19.7 (Write C++ Statements) Use the `std::format` function to write a statement for each of the following:

- a) Print integer 40000 left aligned in a 15-digit field.
- b) Print 200 with and without a sign.
- c) Print the decimal value 100 in hexadecimal form preceded by 0x.
- d) Print 1.234 in a 9-digit field with preceding zeros.

19.8 (Inputting Decimal, Octal and Hexadecimal Values) Write a program to test the inputting of integer values in decimal, octal and hexadecimal formats. Output each integer read by the program in all three formats. Test the program with the following input data: 10, 010, 0x10.

19.9 (Printing Pointer Values as Integers) Write a program that prints pointer values using casts to all the integer data types. Which ones print strange values? Which ones cause errors?

19.10 (Printing with Field Widths) Write a program to test the results of printing the integer value 12345 and the floating-point value 1.2345 in various-sized fields. Do not use the `std::format` function.

19.11 (Printing with Field Widths Using std::format) Using `std::format`, write a program to test the results of printing the integer value 12345 and the floating-point value 1.2345 in various-sized fields. Do not use the `std::format` function.

19.12 (Rounding) Write a program that prints the value 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth. Do not use the `std::format` function.

19.13 (Rounding with std::format) Using `std::format`, write a program that prints the value 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

19.14 (Displaying Fahrenheit and Celsius Temperatures) Write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Use the following formula to perform the calculation:

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

Display the output in two right-aligned columns, and display the Celsius temperatures preceded by a sign for both positive and negative values. Do not use the `std::format` function.

19.15 (Displaying Fahrenheit and Celsius Temperatures with std::format) Using `std::format`, write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Use the following formula to perform the calculation:

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

Display the output in two right-aligned columns, and display the Celsius temperatures preceded by a sign for both positive and negative values.

19.16 (Displaying a Table of ASCII Values) Write a program that uses a `for` statement to print a table of ASCII values for the characters in the ASCII character set from 33 to 126. The program should print the decimal value, octal value, hexadecimal value and character value for each character. Use the stream manipulators `dec`, `oct` and `hex` to print the integer values. Do not use the `std::format` function.

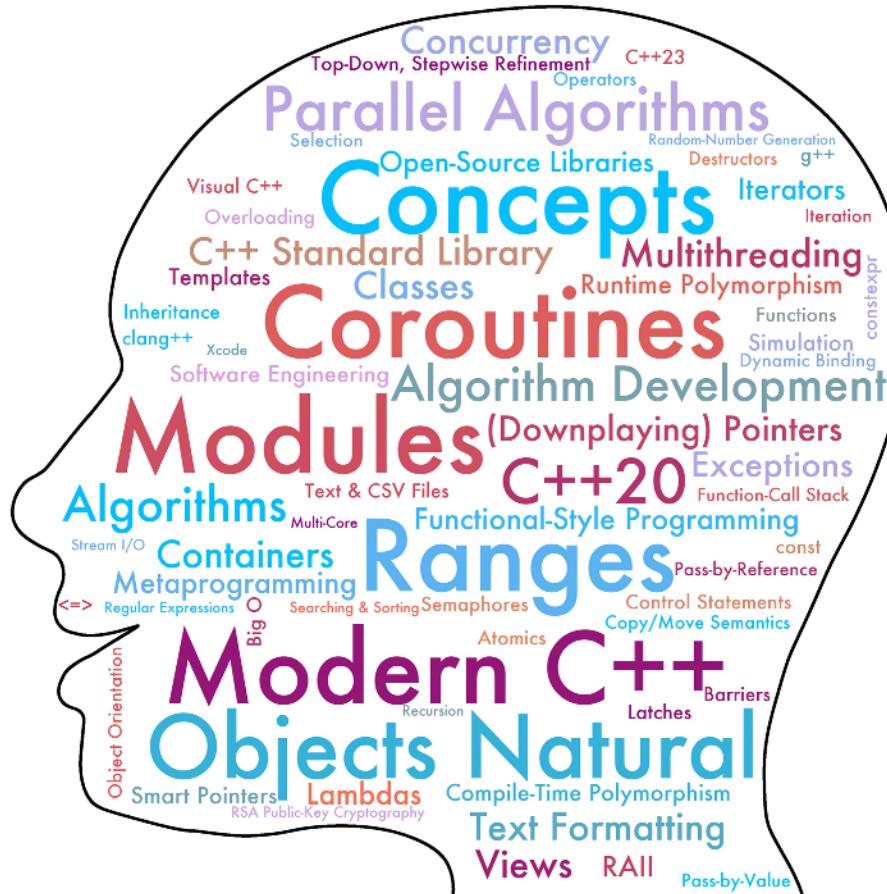
19.17 (Displaying a Table of ASCII Values with `std::format`) Write a program that uses a `for` statement to print a table of ASCII values for the characters in the ASCII character set from 33 to 126. The program should print the decimal value, octal value, hexadecimal value and character value for each character. Use the stream manipulators `dec`, `oct` and `hex` to print the integer values. Do not use the `std::format` function.

19.18 (String-Terminating Null Character) Write a program to show that the `getline` and three-argument `get istream` member functions both end the input string with a string-terminating null character. Also, show that `get` leaves the delimiter character on the input stream, whereas `getline` extracts the delimiter character and discards it. What happens to the unread characters in the stream?

This page intentionally left blank

20

Other Topics and a Look Toward the Future of C++



Objectives

In this chapter, you'll:

- Determine an object's type at runtime.
- Inherit base-class constructors.
- Learn more advanced runtime polymorphism techniques.
- Use multiple inheritance.
- Understand storage classes and storage duration.
- Use `mutable` members in `const` objects.
- Use namespaces to ensure that identifiers are unique.
- Use operator keywords in place of operator symbols.
- Use smart pointers to manage dynamic memory for shared objects.
- Determine types at compile time with `decltype`.
- Use `[[nodiscard]]` to indicate a function's return value should not be ignored.
- Learn some key C++23 features.

Outline

20.1	Introduction	20.9.2	Accessing namespace Members with Qualified Names
20.2	<code>shared_ptr</code> and <code>weak_ptr</code> Smart Pointers	20.9.3	using Directives Should Not Be Placed in Headers
20.2.1	Reference Counted <code>shared_ptr</code>	20.9.4	Nested Namespaces
20.2.2	<code>weak_ptr</code> : <code>shared_ptr</code> Observer	20.9.5	Aliases for namespace Names
20.3	Runtime Polymorphism with <code>std::variant</code> and <code>std::visit</code>	20.10	Storage Classes and Storage Duration
20.4	<code>protected</code> Class Members: A Deeper Look	20.10.1	Storage Duration
20.5	Non-Virtual Interface (NVI) Idiom	20.10.2	Local Variables and Automatic Storage Duration
20.6	Inheriting Base-Class Constructors	20.10.3	Static Storage Duration
20.7	Multiple Inheritance	20.10.4	<code>mutable</code> Class Members
20.7.1	Diamond Inheritance	20.11	Operator Keywords
20.7.2	Eliminating Duplicate Subobjects with virtual Base-Class Inheritance	20.13	<code>decltype</code> Operator
20.8	<code>public</code> , <code>protected</code> and <code>private</code> Inheritance	20.13	Trailing Return Types for Functions
20.9	namespaces: A Deeper Look	20.15	<code>[[nodiscard]]</code> Attribute
20.9.1	Defining namespaces	20.15	Some Key C++23 Features
		20.16	Wrap-Up

20.1 Introduction

This chapter continues our discussion of inheritance and runtime polymorphism that began in Chapter 10, presents additional C++ features and introduces some key C++23 features. In this chapter, we:

- use the **shared_ptr** and **weak_ptr** smart pointers to manage shared dynamically allocated objects.
- implement **runtime polymorphism for objects of classes unrelated by inheritance** using the `std::variant` class template and the `std::visit` function. Invoking common functionality on objects of unrelated types is called **duck typing**.
- demonstrate a runtime polymorphism approach called the **non-virtual interface idiom (NVI)** in which each base-class function serves only one purpose—either as a function client code can call to perform a task or as a function that derived classes can customize. You’ll see that the customizable functions are internal implementation details of the classes, making systems easier to maintain and evolve without requiring code changes in client applications.
- show that a derived class can **inherit its base-class constructors** rather than defining its own that perform the same task(s).
- demonstrate **multiple inheritance**, which enables a derived class to inherit the members of two or more base classes. We discuss potential problems with multiple inheritance and how **virtual inheritance** can solve them.
- continue discussing **namespaces** (introduced in Section 16.6), which help ensure that every identifier in a program has a unique name, thus avoiding name conflicts (for example, between your code and library code). These are crucial for large software projects.

- discuss the **storage classes** and **storage duration**, which together determine an object’s lifetime.
- introduce **operator keywords** for programmers with keyboards that do not support certain characters used in operator symbols, such as !, &, ^, ~ and |.
- introduce the compile-time **`decltype`** operator to deduce types from expressions, typically used in **template metaprogramming** (Section 15.13).
- revisit **trailing return types for functions** (introduced in Section 14.3), typically used in more complex function-template definitions.
- use the **`[[nodiscard]]`** attribute to indicate that a function’s return value should not be ignored, enabling compilers to warn you when the return value is not used in your program. We also show C++20’s **`[[nodiscard]]` enhancement**, which includes a string that the compiler will display as the reason the function’s caller should not ignore the return value.
- introduce some key C++23 features.

20.2 `shared_ptr` and `weak_ptr` Smart Pointers

Sections 11.4–11.5 introduced dynamic memory allocation and smart pointers. You learned that many common bugs in C++ code are related to pointers and dynamically allocated memory and that smart pointers help you avoid such errors by providing additional functionality that strengthens memory allocation and deallocation. Smart pointers also help you write **exception-safe code** (introduced in Section 12.3). If a program throws an exception before `delete` has been called on a pointer to dynamically allocated memory, a memory leak occurs. On the other hand, **smart pointers use resource acquisition is initialization (RAII; Section 11.5)**. A smart pointer is an object, so when a smart pointer goes out of scope for any reason, including an exception, the smart pointer’s destructor will still be called, helping avoid memory leaks. Section 11.5 introduced the `unique_ptr` class template. Here, we discuss `shared_ptr` and `weak_ptr`.

AS
Err

20.2.1 Reference Counted `shared_ptr`

A `shared_ptr` (header `<memory>`) holds an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared throughout a program.¹ You can have many `shared_ptr`s to the same resource. As the type name implies, `shared_ptr`s share a resource—if you change the resource with one `shared_ptr`, the changes also will be “seen” by the other `shared_ptr`s. The internal pointer is deleted when the last `shared_ptr` to the resource is destroyed. Internally, `shared_ptr`s use **reference counting** to determine how many `shared_ptr`s point to the resource. Each time you create a new `shared_ptr` to a resource, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted, and the memory is released.

1. “`std::shared_ptr`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/memory/shared_ptr.

shared_ptrs vs. unique_ptrs

The C++ Core Guidelines indicate that you should prefer a `unique_ptr` over a `shared_ptr` if there will be only one owner of a given resource at a time.² 

Example Using shared_ptr

Fig. 20.1 defines a simple `Book` class (lines 12–17) with a `string` representing the Book's title. `Book`'s destructor (line 15) displays a message indicating which `Book` object is being destroyed. We use this class to demonstrate the basic functionality of `shared_ptr`.

```

1 // fig20_01.cpp
2 // Demonstrate shared_ptrs.
3 #include <algorithm>
4 #include <format>
5 #include <iostream>
6 #include <memory>
7 #include <string>
8 #include <string_view>
9 #include <vector>
10
11 // class Book
12 class Book {
13 public:
14     explicit Book(std::string_view bookTitle) : title{bookTitle} {}
15     ~Book() {std::cout << std::format("Destroying Book: {}\\n", title);}
16     std::string title; // title of the Book
17 };
18
19 // a custom delete function for a pointer to a Book
20 void deleteBook(Book* book) {
21     std::cout << "Custom deleter for a Book, ";
22     delete book; // delete the Book pointer
23 }
24
25 // compare the titles of two Books for sorting
26 bool compareTitles(
27     std::shared_ptr<Book> ptr1, std::shared_ptr<Book> ptr2) {
28     return (ptr1->title < ptr2->title);
29 }
30
31 int main() {
32     // create a shared_ptr to a Book and display the reference count
33     std::shared_ptr<Book> bookPtr{
34         std::make_shared<Book>("C++ How to Program")};
35     std::cout << std::format("Reference count for Book {} is: {}\\n",
36                             bookPtr->title, bookPtr.use_count());
37 }
```

Fig. 20.1 | `shared_ptr` example program. (Part 1 of 3.)

2. C++ Core Guidelines, “F.27: Use a `shared_ptr<T>` to share ownership.” Accessed April 18, 2023. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-shared_ptr.

```

38 // create another shared_ptr to the Book and display reference count
39 std::shared_ptr<Book> bookPtr2{bookPtr};
40 std::cout << std::format("Reference count for Book {} is: {}\\n",
41     bookPtr->title, bookPtr.use_count());
42
43 // change the Book's title and access it from both pointers
44 bookPtr2->title = "Java How to Program";
45 std::cout << std::format(
46     "Updated Book title:\\nbookPtr: {}\\nbookPtr2: {}\\n",
47     bookPtr->title, bookPtr2->title);
48
49 // create a std::vector of shared_ptrs to Books (BookPtrs)
50 std::vector<std::shared_ptr<Book>> books{
51     std::make_shared<Book>("C How to Program"),
52     std::make_shared<Book>("Intro to Python"),
53     std::make_shared<Book>("C# How to Program"),
54     std::make_shared<Book>("C++ How to Program");
55
56 // print the Books in the vector
57 std::cout << "\\nBooks before sorting:\\n";
58 for (auto book : books) {
59     std::cout << book->title << "\\n";
60 }
61
62 // sort the vector by Book title and print the sorted vector
63 std::sort(books.begin(), books.end(), compareTitles);
64 std::cout << "\\nBooks after sorting:\\n";
65 for (auto book : books) {
66     std::cout << book->title << "\\n";
67 }
68
69 // create a shared_ptr with a custom deleter
70 std::cout << "\\nshared_ptr with a custom deleter.\\n";
71 std::shared_ptr<Book> bookPtr3{
72     new Book("Android How to Program"), deleteBook};
73 bookPtr3.reset(); // release the Book this shared_ptr manages
74
75 // shared_ptrs are going out of scope
76 std::cout << "\\nEnd of main: shared_ptr objects going out of scope.\\n";
77 }

```

Reference count for Book C++ How to Program is: 1

Reference count for Book C++ How to Program is: 2

Updated Book title:

bookPtr: Java How to Program

bookPtr2: Java How to Program

Books before sorting:

C How to Program

Intro to Python

C# How to Program

C++ How to Program

Fig. 20.1 | shared_ptr example program. (Part 2 of 3.)

```

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
Intro to Python

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Android How to Program

End of main: shared_ptr objects going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: Intro to Python
Destroying Book: Java How to Program

```

Fig. 20.1 | `shared_ptr` example program. (Part 3 of 3.)

Creating `shared_ptrs`

The program uses `shared_ptr`s to manage several `Book` instances. Lines 33–34 use the standard library function `make_shared` to create a `shared_ptr` to a dynamically allocated

CG (●) `Book` object with the title "C++ How to Program".³ This is the recommended way to create a `shared_ptr`.⁴ You can simplify this statement by declaring `bookPtr` with `auto`. The `shared_ptr` manages the `Book` object and initially sets its reference count to 1—one `shared_ptr` currently refers to the dynamically allocated object. If a new `shared_ptr` is initialized with an existing one, they both share ownership of the resource and the reference count increases by 1. If you make multiple `shared_ptr`s with the same pointer, the `shared_ptr`s won't acknowledge each other, and the reference count will be wrong. When the `shared_ptr`s are destroyed, they decrease the reference count by one. The `shared_ptr` destructor call that decrements the shared resource's reference count to 0 also deletes the resource.

Manipulating `shared_ptrs`

Lines 35–36 display the `Book`'s `title` and the number of `shared_ptr`s referencing that object—that is, the current reference count. We use the `->` operator to access the `Book`'s data member `title`—`shared_ptr`'s overload the pointer operators `->` and `*`, so you can use them like raw pointers. The `shared_ptr` member function `use_count` returns the reference count.

Line 39 creates another `shared_ptr` to the same `Book` using the `shared_ptr` constructor that receives a `shared_ptr` argument—this increments the `Book`'s reference count by one. You also can use `shared_ptr`'s assignment operator (`=`) to create a `shared_ptr` to the same resource. Lines 40–41 display the `Book`'s `title` and reference count again to show that the reference count increased by one when we created the second `shared_ptr`.

As mentioned earlier, changes made to the resource of a `shared_ptr` are “seen” by all `shared_ptr`s to that resource. Line 44 uses `bookPtr2` to change the `Book`'s `title`, then

-
3. “`std::make_shared`, `std::make_shared_for_overwrite`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared.
 4. C++ Core Guidelines, “R.22: Use `make_shared()` to make `shared_ptr`s.” Accessed April 18, 2023. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-make_shared.

lines 45–47 display the Book's title using bookPtr and bookPtr2 to show that bookPtr "sees" the change.

Manipulating shared_ptrs in an STL Container

Next, we demonstrate using shared_ptrs in an STL container. Lines 50–54 create a vector of `shared_ptr<Book>` objects and add four elements, using `make_shared` to dynamically allocate each Book and return a `shared_ptr` to it. Lines 58–60 display the vector's initial contents. Then, line 63 sorts the Books by title, using the function `compareTitles` (lines 26–29) to customize how the `sort` algorithm compares Book objects—in this case, by comparing the Books' titles. Lines 65–67 display the vector's sorted contents.

Customizing How Shared Resources Are Destroyed with a Custom Deleter

`shared_ptr`s also allow you to determine how resources will be destroyed. For most dynamically allocated objects, `delete` is used. However, some resources require more complex cleanup. In that case, you can supply a custom `deleter` function (a function pointer, lambda or function object) to the `shared_ptr` constructor. The deleter specifies how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the `shared_ptr` calls the custom deleter function.

Lines 71–72 create a `shared_ptr` with the custom deleter function `deleteBook` (lines 20–23), passed as the second argument to the `shared_ptr` constructor. A **custom deleter function must take one argument of the `shared_ptr`'s internal pointer type**. When the last `shared_ptr` for a given Book object is destroyed, the `shared_ptr` passes its internal `Book*` to the custom deleter, which must delete the managed resource. In this example, `deleteBook` displays a message showing that the custom deleter was called, then deletes the dynamically allocated Book object.

Resetting a shared_ptr

Line 73 calls the `shared_ptr` member function `reset`, which sets the `shared_ptr` to `nullptr`, so it no longer points to a shared resource. Since `bookPtr3` was the only `shared_ptr` to the Book object created in lines 71–72, the `shared_ptr` calls its custom deleter to release the resource. Other versions of `reset` allow you to pass a new resource to manage.

shared_ptrs Are Destroyed When They Go Out of Scope

When `main` terminates, all the `shared_ptr`s and the `vector` in this example go out of scope and are destroyed. Before the `vector` is destroyed, its `shared_ptr` elements are destroyed. The output shows that each Book object is destroyed automatically by the `shared_ptr`s.

20.2.2 weak_ptr: shared_ptr Observer

A `weak_ptr` points to the resource managed by a `shared_ptr` without assuming any responsibility for it.⁵ A `shared_ptr`'s reference count does not increase when a `weak_ptr` references the shared resource. Thus, a `shared_ptr`'s resource can be deleted even if there are still `weak_ptr`s pointing to it. When the last `shared_ptr` to a resource is destroyed, any remaining `weak_ptr`s are set to `nullptr`. The C++ Core Guidelines say to use



5. “`std::weak_ptr`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/memory/weak_ptr.

Err  `weak_ptrs` “to break cycles of `shared_ptrs`.⁶ As we’ll demonstrate later in this section, this helps avoid memory leaks caused by circular references.

A `weak_ptr` cannot directly access the resource it points to—you must create a `shared_ptr` from the `weak_ptr` to access the resource. This enables a program to determine whether the resource still exists. There are two ways to create a `shared_ptr` from a `weak_ptr`:

Err 

- You can pass the `weak_ptr` to the `shared_ptr` constructor, which creates a `shared_ptr` to the resource pointed to by the `weak_ptr` and properly increases the reference count. If the resource has already been deleted, the `shared_ptr` constructor will throw a `bad_weak_ptr` exception.
- You can also call the `weak_ptr` member function `lock`, which returns a `shared_ptr` to the `weak_ptr`’s resource.⁷ If the `weak_ptr` does not currently point to a resource, `lock` returns an empty `shared_ptr` (i.e., a `shared_ptr` with its internal pointer set to `nullptr`). You should use `lock` if an empty `shared_ptr` isn’t considered an error in the context of your application.

Err 

One typical use-case of `weak_ptrs` is in **circularly referential data**—a situation in which two objects refer to each other internally, as the example in Figs. 20.2–20.5 demonstrates. Another use-case is avoiding **dangling pointers**⁸—that is, pointers to dynamically allocated objects that are subsequently deleted. Using such a pointer is undefined behavior that often crashes a program. Because a `weak_ptr` cannot directly access the resource it points to, the resource can be deleted. If the program subsequently attempts to use the `weak_ptr`’s `lock` function to get a `shared_ptr`, `lock` returns `nullptr` to indicate that the resource no longer exists.

Class Definitions for Classes Author and Book

We’ll demonstrate `weak_ptrs` using classes `Author` and `Book` (Figs. 20.2–20.5). Each class contains pointers to an object of the other class. We also use `public` data members here only to simplify the code. This creates a **circular type reference** between the class definitions:

- class `Author` defines a `weak_ptr` and a `shared_ptr` to a `Book` (lines 20–21 in Fig. 20.2), and
- class `Book` defines a `weak_ptr` and a `shared_ptr` to an `Author` (lines 20–21 in Fig. 20.3).

To declare these pointers, each class needs to know that the other exists, which you usually accomplish by including the corresponding class’s header. However, we cannot do that here because class `Author` depends on class `Book`, which, in turn, depends on class `Author`. Including `Book.h` in `Author.h` and including `Author.h` in `Book.h` causes compilation errors. Interestingly, declaring a pointer does not require a complete type definition—it simply requires knowing that the type exists. This is the purpose of the **forward class declarations** in line 8 of Figs. 20.2 and 20.3. To use a pointer to interact with an object, the

-
6. C++ Core Guidelines, “R.24: Use `std::weak_ptr` to break cycles of `shared_ptrs`.” Accessed April 18, 2023. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-weak_ptr.
 7. “`std::weak_ptr::lock`.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/memory/weak_ptr/lock.
 8. “When is `std::weak_ptr` useful?” Accessed April 18, 2023. <https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful>.

full class definition must be available, as you'll see in the .cpp files for classes Author (Fig. 20.4) and Book (Fig. 20.5).

```
1 // Fig. 20.2: Author.h
2 // Author class definition.
3 #pragma once
4 #include <memory>
5 #include <string>
6 #include <string_view>
7
8 class Book; // forward declaration of class Book
9
10 // Author class definition
11 class Author {
12 public:
13     explicit Author(std::string_view authorName);
14     ~Author();
15
16     // print the title of the Book this Author wrote
17     void printBookTitle();
18
19     std::string name; // name of the Author
20     std::weak_ptr<Book> weakBookPtr; // Book the Author wrote
21     std::shared_ptr<Book> sharedBookPtr; // Book the Author wrote
22 };
```

Fig. 20.2 | Author class definition.

```
1 // Fig. 20.3: Book.h
2 // Book class definition.
3 #pragma once
4 #include <memory>
5 #include <string>
6 #include <string_view>
7
8 class Author; // forward declaration of class Author
9
10 // Book class definition
11 class Book {
12 public:
13     explicit Book(std::string_view bookTitle);
14     ~Book();
15
16     // print the name of this Book's Author
17     void printAuthorName();
18
19     std::string title; // title of the Book
20     std::weak_ptr<Author> weakAuthorPtr; // Author of the Book
21     std::shared_ptr<Author> sharedAuthorPtr; // Author of the Book
22 };
```

Fig. 20.3 | Book class definition.

 Using classes `Author` and `Book`, we'll show that setting an `Author` object's `shared_ptr` to point to a `Book` object and setting that `Book` object's `shared_ptr` to point back to the `Author` object creates a memory leak. Then, we'll show how we can use the `weak_ptrs` to fix this problem.

Member Function Definitions for Classes `Author` and `Book`

Figures 20.4 and 20.5 define `Author`'s and `Book`'s member functions, respectively. For an `Author` object's member functions to manipulate a corresponding `Book` object through a pointer, class `Author` must know class `Book`'s complete definition. So line 9 in Fig. 20.4 includes `Book.h`. Similarly, for a `Book` object's member functions to manipulate a corresponding `Author` object through a pointer, class `Book` must know class `Author`'s complete definition. So line 8 in Fig. 20.5 includes `Author.h`.

```

1 // Fig. 20.4: Author.cpp
2 // Author member-function definitions.
3 #include <format>
4 #include <iostream>
5 #include <memory>
6 #include <string>
7 #include <string_view>
8 #include "Author.h"
9 #include "Book.h"
10
11 Author::Author(std::string_view authorName) : name(authorName) {}
12
13 Author::~Author() {
14     std::cout << std::format("Destroying Author: {}\n", name);
15 }
16
17 // print the title of the Book this Author wrote
18 void Author::printBookTitle() {
19     // if weakBookPtr.lock() returns a non-empty shared_ptr
20     if (std::shared_ptr<Book> bookPtr{weakBookPtr.lock()}) {
21         // show the reference count increase and print the Book's title
22         std::cout << std::format("Reference count for Book {} is {}\n",
23             bookPtr->title, bookPtr.use_count());
24         std::cout << std::format("Author {} wrote the book {}\n",
25             name, bookPtr->title);
26     }
27     else { // weakBookPtr points to NULL
28         std::cout << "This Author has no Books.\n";
29     }
30 }
```

Fig. 20.4 | Author member-function definitions.

```

1 // Fig. 20.5: Book.cpp
2 // Book member-function definitions.
3 #include <format>
4 #include <iostream>
5 #include <memory>
6 #include <string>
7 #include <string_view>
8 #include "Author.h"
9 #include "Book.h"
10
11 Book::Book(std::string_view bookTitle) : title(bookTitle) {}
12
13 Book::~Book() {
14     std::cout << std::format("Destroying Book: {}\n", title);
15 }
16
17 // print the name of this Book's Author
18 void Book::printAuthorName() {
19     // if weakAuthorPtr.lock() returns a non-empty shared_ptr
20     if (std::shared_ptr<Author> authorPtr{weakAuthorPtr.lock()}) {
21         // show the reference count increase and print the Author's name
22         std::cout << std::format("Reference count for Author {} is {}\n",
23             authorPtr->name, authorPtr.use_count());
24         std::cout << std::format("The book {} was written by {}\n",
25             title, authorPtr->name);
26     }
27     else { // weakAuthorPtr points to NULL
28         std::cout << "This Book has no Author.\n";
29     }
30 }

```

Fig. 20.5 | Book member-function definitions.

Both classes define destructors that display a message to indicate when an object of either class is destroyed (Figs. 20.4 and 20.5, lines 13–15). Class **Author**'s **printBookTitle** function (Fig. 20.4, lines 18–30) displays its Book's **title** and reference count (lines 22–23), then shows that the **Author** wrote that Book (lines 24–25). Similarly, class **Book**'s **printAuthorName** function (Fig. 20.5, lines 18–30) displays its Author's name and reference count (lines 22–23), then shows that the Book was written by that Author (lines 24–25).

Recall that you can't access a resource directly through a **weak_ptr**. For **printBookTitle** and **printAuthorName** to perform the tasks in lines 22–25 of each, these functions must first create a **shared_ptr** from the **weak_ptr** data member by calling the **weak_ptr**'s **lock** function (line 20 in each figure). If the **weak_ptr** does not reference a resource, the **lock** function returns a **shared_ptr** containing **nullptr**, which evaluates to **false** in line 20. Otherwise, the new **shared_ptr** contains a valid pointer to the **weak_ptr**'s resource, the condition in line 20 evaluates to **true**, and lines 22–25 of each figure can access the corresponding resource. Lines 22–23 of each function display the **shared_ptr**'s reference count to show that creating the new **shared_ptr** increased the reference count for the corresponding **Book** (Fig. 20.4) or **Author** (Fig. 20.5). The new **shared_ptr** created in line 20 of each figure is a local object, so it's destroyed when its enclosing block ends. At that point, the corresponding reference count decreases by one.

main Function

Figure 20.6 demonstrates the **memory leak** caused by the **circular reference** between Author and Book. Lines 11–14 create shared_ptrs to an object of each class. Line 17 sets the Book's weak_ptr data member to point to the Author, and line 18 sets the Author's weak_ptr data member to point to the Book. Similarly, lines 21–22 set the shared_ptr data members of each object. The Author and Book objects now reference each other.

```

1 // fig20_06.cpp
2 // Demonstrate use of weak_ptr.
3 #include <format>
4 #include <iostream>
5 #include <memory>
6 #include "Author.h"
7 #include "Book.h"
8
9 int main() {
10     // create a Book and an Author
11     std::shared_ptr<Book> bookPtr(
12         std::make_shared<Book>("C++ How to Program"));
13     std::shared_ptr<Author> authorPtr(
14         std::make_shared<Author>("Deitel & Deitel"));
15
16     // reference the Book and Author to each other
17     bookPtr->weakAuthorPtr = authorPtr;
18     authorPtr->weakBookPtr = bookPtr;
19
20     // set the shared_ptr data members to create the memory leak
21     bookPtr->sharedAuthorPtr = authorPtr;
22     authorPtr->sharedBookPtr = bookPtr;
23
24     // reference count for bookPtr and authorPtr is two
25     std::cout << std::format("Reference count for Book {} is {}\n",
26                             bookPtr->title, bookPtr.use_count());
27     std::cout << std::format("Reference count for Author {} is {}\n\n",
28                             authorPtr->name, authorPtr.use_count());
29
30     // access the cross references to print the data they point to
31     std::cout << "Access Author name and Book title via weak_ptrs:\n";
32     bookPtr->printAuthorName();
33     std::cout << "\n";
34     authorPtr->printBookTitle();
35
36     // reference count for each shared_ptr is two
37     std::cout << std::format("Reference count for Book {} is {}\n",
38                             bookPtr->title, bookPtr.use_count());
39     std::cout << std::format("Reference count for Author {} is {}\n\n",
40                             authorPtr->name, authorPtr.use_count());
41
42     // the shared_ptrs go out of scope, the Book and Author are destroyed
43     std::cout << "End of main. shared_ptrs going out of scope.\n";
44 }
```

Fig. 20.6 | shared_ptrs cause a memory leak in circularly referential data. (Part I of 2.)

```
Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 3
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 3
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

End of main. shared_ptrs going out of scope.
```

Fig. 20.6 | shared_ptrs cause a memory leak in circularly referential data. (Part 2 of 2.)

Next, lines 25–28 display the reference counts for the `shared_ptr`s to show that each object is referenced by two `shared_ptr`s:

- the `Book` object is referenced by the `shared_ptr` in lines 11–12 and by the one in the `Author` object (line 22), and
- the `Author` object is referenced by the `shared_ptr` in lines 13–14 and by the one in the `Book` object (line 21).

Note that the `weak_ptr`s don't affect the reference counts. Line 32 calls the `Book` object's `printAuthorName` function to display the information stored in the `Book`'s `weak_ptr` data member. Line 34 calls the `Author` object's `printBookTitle` function to display the information stored in the `Author`'s `weak_ptr` data member. Each of these functions also displays the fact that another `shared_ptr` was created during the function call increasing the `Book`'s and `Author`'s reference counts to 3. Finally, lines 37–40 display the `Book`'s and `Author`'s reference counts again to show that the additional `shared_ptr`s created in the `printAuthorName` and `printBookTitle` were destroyed when the functions finished executing, decreasing each object's reference count to 2. Then, `main` terminates.

Memory Leak

At the end of `main`, the `shared_ptr`s to the instances of `Author` and `Book` we created go out of scope and are destroyed, but notice that the output doesn't show the destructors for classes `Author` and `Book`. This program has a **memory leak**—the `Author` and `Book` objects aren't destroyed because of their `shared_ptr` data members. When `bookPtr` is destroyed at the end of the `main` function, the reference count for the object of class `Book` becomes one—the object of `Author` still has a `shared_ptr` to the `Book` object, so it isn't deleted. When `authorPtr` goes out of scope and is destroyed, the reference count for the object of class `Author` also becomes one—the `Book` object still has a `shared_ptr` to the `Author` object. Neither object is deleted because the reference count for each is still one.



Fixing the Memory Leak

Now, comment out lines 21–22 in `main` by placing `//` at the beginning of each line. This prevents the code from setting the `shared_ptr` data members for classes `Author` and `Book`. Recompile the code and run the program again to produce the following output. We bolded the last two lines showing the `Author` and `Book` objects were properly destroyed.

```

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 2
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 2
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

End of main. shared_ptrs going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program

```

Notice that the initial reference count for each object is now 1 instead of 2 because we don't set the `shared_ptr` data members. The last two lines of the output show that the `Author` and `Book` objects were destroyed at the end of `main`. We eliminated the memory leak by using only the `weak_ptr` data members, which don't affect the reference count but still allow us to access the resource when we need it by creating a temporary `shared_ptr` to the resource. When the `shared_ptr`s we created in `main` are destroyed, their reference counts become 0, and the instances of classes `Author` and `Book` are deleted properly.

20.3 Runtime Polymorphism with `std::variant` and `std::visit`

So far, our runtime polymorphism examples used implementation inheritance or interface inheritance—both 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 the class template `std::variant` and the standard-library function `std::visit` (from header `<variant>`).^{9,10,11,12} 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.



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."*¹³

-
9. Nevin Liber, "The Many Variants of `std::variant`," YouTube Video, June 16, 2019. Accessed April 18, 2023. <https://www.youtube.com/watch?v=JUxhwf7gYLg>.
 10. "`std::variant`." Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/utility/variant>.
 11. Bartłomiej Filipek, "Runtime Polymorphism with `std::variant` and `std::visit`," November 2, 2020. Accessed April 18, 2023. <https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html>.
 12. Bartłomiej Filipek, "Everything You Need to Know About `std::variant` from C++17," June 4, 2018. Accessed April 18, 2023. <https://www.bfilipek.com/2018/06/variant.html>.
 13. "Duck Typing." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Duck_typing.

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, as you'll see in the next example.

To demonstrate runtime polymorphism using duck typing with `std::variant`, we'll reimplement our interface inheritance example from Section 10.11. The classes used here are nearly identical, so we'll point out only the differences.

Compensation Model Salaried

Class `Salaried` (Figs. 20.7–20.8) defines the compensation model for an `Employee` who gets paid a fixed salary. The only difference between this class and the one in Section 10.11 is that this `Salaried` is not a derived class. So, its `earnings` and `toString` member functions do not override base-class `virtual` functions.

```

1 // Fig. 20.7: 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. 20.7 | Salaried compensation model.

```

1 // Fig. 20.8: Salaried.cpp
2 // Salaried compensation model member-function definitions.
3 #include <format>
4 #include <stdexcept>
5 #include "Salaried.h" // class definition
6
7 // constructor
8 Salaried::Salaried(double salary) : m_salary{salary} {
9     if (m_salary < 0.0) {
10         throw std::invalid_argument("Weekly salary must be >= 0.0");
11     }
12 }
13
14 // calculate earnings
15 double Salaried::earnings() const {return m_salary;}
16
17 // return string containing Salaried compensation model information
18 std::string Salaried::toString() const {
19     return std::format("salary: ${:.2f}", m_salary);
20 }

```

Fig. 20.8 | Salaried compensation model member-function definitions.

Compensation Model Commission

Class `Commission` (Figs. 20.9–20.10) 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 10.11.

```

1 // Fig. 20.9: 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 };

```

Fig. 20.9 | Commission compensation model.

```

1 // Fig. 20.10: Commission.cpp
2 // Commission member-function definitions.
3 #include <format>
4 #include <stdexcept>
5 #include "Commission.h" // class definition
6
7 // constructor
8 Commission::Commission(double grossSales, double commissionRate)
9     : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
10
11     if (m_grossSales < 0.0) {
12         throw std::invalid_argument("Gross sales must be >= 0.0");
13     }
14
15     if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
16         throw std::invalid_argument(
17             "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

```

Fig. 20.10 | Commission member-function definitions. (Part 1 of 2.)

```

26 // return string containing Commission information
27 std::string Commission::toString() const {
28     return std::format("gross sales: {:.2f}; commission rate: {:.2f}",
29                         m_grossSales, m_commissionRate);
30 }
```

Fig. 20.10 | Commission member-function definitions. (Part 2 of 2.)

Employee Class Definition

As in Section 10.11, each `Employee` (Fig. 20.11) 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:

```
using CompensationModel = std::variant<Commission, Salaried>;
```

defines the alias `CompensationModel` for the `std::variant` type:

```
std::variant<Commission, Salaried>
```

Such `using` declarations are **alias declarations**. They enable you to create convenient names for complex types—such as a `std::variant` type that potentially could have 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.



```

1 // Fig. 20.11: 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 }
```

Fig. 20.11 | An `Employee` “has a” `CompensationModel`.

Type-Safe union

A `union`¹⁴ is a region of memory that, over time, can contain objects of various types. 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`.¹⁵

Employee Constructor and `setCompensationModel` Member Function

Class `Employee`'s member functions (Fig. 20.12) perform the same tasks as in Fig. 10.21 with several modifications. The `Employee` class's constructor (lines 8–9) and `setCompensationModel` member function (lines 12–14) each receive a `CompensationModel` object. Unlike the composition-and-dependency-injection approach, the `std::variant` object stores an actual object, not a pointer to one.

```

1 // Fig. 20.12: Employee.cpp
2 // Class Employee member-function definitions.
3 #include <format>
4 #include <string>
5 #include "Employee.h"
6
7 // constructor
8 Employee::Employee(std::string_view name, CompensationModel model)
9     : m_name{name}, m_model{model} {}
10
11 // change the Employee's CompensationModel
12 void Employee::setCompensationModel(CompensationModel model) {
13     m_model = model;
14 }
15
16 // return the Employee's earnings
17 double Employee::earnings() const {
18     auto getEarnings{[](const auto& model){return model.earnings();}};
19     return std::visit(getEarnings, m_model);
20 }
21
22 // return string representation of an Employee object
23 std::string Employee::toString() const {
24     auto getString{[](const auto& model){return model.toString();}};
25     return std::format("{}\n{}", m_name, std::visit(getString, m_model));
26 }
```

Fig. 20.12 | Class `Employee` member-function definitions.

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

-
- 14. “Union Declaration.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/union>.
 - 15. “`std::variant`.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/utility/variant>.

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 18–19 in member-function `earnings`:

```
auto getEarnings{}[](const auto& model){return model.earnings();};  
return std::visit(getEarnings, m_model);
```

Line 18 defines the variable `getEarnings` and initializes it with a generic lambda expression 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 19 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 24 in `Employee`'s `toString` member function creates a lambda expression that returns the result of calling some object's `toString` member function. Line 25 calls `std::visit` to pass `m_model` to the lambda expression, which returns `m_model`'s `toString` result.

Testing Runtime Polymorphism with `std::variant` and `std::visit`

In `main`, lines 11–12 create two `Employee` objects—the first stores a `Salaried` object in its `std::variant` and the second stores a `Commission` object. The expression

```
Salaried{800.0}
```

in line 11 creates a `Salaried` object, which is immediately used to initialize the `Employee`'s `CompensationModel`. Similarly, the expression

```
Commission{10000.0, .06}
```

in line 12 creates a `Commission` object, which is immediately used to initialize the `Employee`'s `CompensationModel`. Line 15 creates a vector of `Employees`, then lines 18–21 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 10.11.

```
1 // fig20_13.cpp  
2 // Processing Employees with various compensation models.  
3 #include <iostream>  
4 #include <vector>  
5 #include <variant>  
6 #include "Employee.h"  
7 #include "Salaried.h"  
8 #include "Commission.h"  
9  
10 int main() {  
11     Employee salariedEmployee{"Pierre Simon", Salaried{800.0}};  
12     Employee commissionEmployee{"Sierra Dembo", Commission{10000.0, .06}};  
13  
14     // create and initialize vector of Employees  
15     std::vector employees{salariedEmployee, commissionEmployee};
```

Fig. 20.13 | Processing Employees with various compensation models.

```

16
17     // print each Employee's information and earnings
18     for (const Employee& employee : employees) {
19         std::cout << std::format("{}\nearned: ${:.2f}\n\n",
20             employee.toString(), employee.earnings());
21     }
22 }
```

```

Pierre Simon
salary: $800.00
earned: $800.00

Sierra Dembo
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 20.13 | Processing Employees with various compensation models.

20.4 protected Class Members: A Deeper Look

Chapter 9 introduced the access specifiers `public` and `private`. A base class's `public` members are accessible within its class, and anywhere the program can access 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`. 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



`protected` data members can 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 depend on the base class's data. Derived classes should depend only on non-private base-class 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 problem**¹⁶ and is a key reason the C++ Core Guidelines recommend avoiding **protected** data.¹⁷ You should be able to change a base class without requiring changes in its derived classes.

 CG

Using **private** data members is better for good software engineering. Declaring base-class data members **private** lets you change the base-class implementation without requiring changes in derived-class implementations. This makes your code easier to maintain, modify and debug.

 SE

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 inheriting from that derived class could access the data members as well, and the benefits of information hiding would be lost.

 SE

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 is to enable a derived class to override the base-class **protected virtual** function and call the original base-class version from the derived class's implementation.¹⁸ We used this capability in the next section.

 SE

20.5 Non-Virtual Interface (NVI) Idiom

The **non-virtual interface (NVI) idiom**^{19,20}—first proposed by Herb Sutter (Convener of the ISO C++ standards committee) in his "Virtuality" paper²¹—is another way to implement runtime polymorphism using class hierarchies. This idiom focuses on the good software-engineering practice of separating a class's interface from its implementation. In Chapter 10's runtime polymorphism examples, each **public virtual** function actually served two purposes:

 SE

16. "Fragile Base Class." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Fragile_base_class.
17. C++ Core Guidelines, "C.133: Avoid **protected** Data." Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-protected>.
18. Herb Sutter, "Virtuality," September 2001. Accessed April 18, 2023. <http://www.gotw.ca/publications/mill18.htm>.
19. "Non-Virtual Interface (NVI) Pattern." Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. https://en.wikipedia.org/wiki/Non-virtual_interface_pattern.
20. Marius Bancila, *Modern C++ Programming Cookbook: Master C++ Core Language and Standard Library Features, with over 100 Recipes, Updated to C++20*. Birmingham: Packt Publishing, 2020, pp. 562–567.
21. Herb Sutter, "Virtuality," *C/C++ Users Journal*, vol. 19, no. 9, September 2001. Accessed April 18, 2023. <http://www.gotw.ca/publications/mill18.htm>.

1. it described part of a class's interface, indicating how client code could interact with an object of that class, and
2. it described part of a class's implementation, enabling derived classes to customize behavior (an implementation detail) by overriding the `virtual` function.

So each public `virtual` function serves as both part of the class's interface and its implementation.

For that reason, Sutter says each function should serve only one of these two purposes. Interestingly, a derived class can override its base class's `private virtual` functions. So, he recommends the following approach to implementing member functions in a class hierarchy:

- Make each `virtual` function `private`, so it is an implementation detail, not part of the class's `public` interface. Doing so ensures that its only purpose is enabling derived classes to customize the base-class behavior by overriding the `virtual` function. You can make a `virtual` function `protected` if each derived class needs to call the base-class `virtual` function as part of its overridden function definition.
- Use a `public non-virtual` function in the base class to invoke that `virtual` function. That `public non-virtual` function will be inherited by all the derived classes and cannot be overridden, so it serves only one purpose—a function the client code calls to perform a task. When the `public non-virtual` function is called on a derived class object, internally it will call the derived class's overridden version of the base-class `virtual` function.

Refactoring Class Employee for the NVI Idiom

Per Sutter's guidelines, let's refactor our `Employee` class (Figs. 10.11–10.12). Most of the code is identical, so we focus on the following key changes in Fig. 20.14:

- `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—customization points for derived classes—will now be implemented via new member functions.
- We added the `protected virtual` function `getString` (line 18) as a customization point for derived classes. This function is called by the non-`virtual` `toString` function. A base class's `protected` members are accessible to its derived classes. As you'll see, our derived classes' `getString` functions will override and call class `Employee`'s `getString`.
- We added the `private pure virtual` function `getPay` (line 21 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.

```

1 // Fig. 20.14: 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};

```

Fig. 20.14 | Employee abstract base class.

Figure 20.15 contains several key changes in class `Employee`'s member-function implementations:

- Member function `earnings` (line 17) now provides a **concrete implementation** that returns the result of calling the `private` `pure virtual` function `getPay`. Even though `getPay` is not yet implemented, this call is allowed because, at execution time, `earnings` will be called on an object of a concrete derived class of `Employee`.
- Member function `toString` (line 20) now returns the result of calling the `protected virtual` function `getString`.
- We now define the new `protected` member function `getString` (lines 23–25) to specify an `Employee`'s default `string` representation containing an `Employee`'s name. This function is `protected` so derived classes can override it and call it to get the base-class part of the derived-class `string` representations.

```

1 // Fig. 20.15: Employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include <format>
5 #include "Employee.h" // Employee class definition
6
7 // constructor
8 Employee::Employee(std::string_view name) : m_name{name} {} // empty body
9

```

Fig. 20.15 | Abstract-base-class `Employee` member-function definitions. (Part 1 of 2.)

```

10 // set name
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // get name
14 std::string Employee::getName() const {return m_name;}
15
16 // public non-virtual function; returns Employee's earnings
17 double Employee::earnings() const {return getPay();}
18
19 // public non-virtual function; returns Employee's string representation
20 std::string Employee::toString() const {return getString();}
21
22 // protected virtual function that derived classes can override and call
23 std::string Employee::getString() const {
24     return std::format("name: {}", getName());
25 }
```

Fig. 20.15 | Abstract-base-class `Employee` member-function definitions. (Part 2 of 2.)

Updated Class `SalariedEmployee`

Our refactored `SalariedEmployee` class (Figs. 20.16–20.17) has several key changes:

- The class’s header (Fig. 20.16) no longer contains prototypes for the `earnings` and `toString` member functions. These **public non-virtual base-class functions are now inherited from `Employee`.**
- The class’s `private` section now declares overrides for the base class’s `private` pure virtual function `getPay` and protected virtual function `getString` (Fig. 20.16, lines 19–20). Function `getString` is **private** in `SalariedEmployee` because this class is **final**, so no other classes can derive from it.
- The class’s member-function implementations (Fig. 20.17) now include overridden function `getPay` (line 27) to return the salary and overridden function `getString` (lines 30–33) to get a `SalariedEmployee`’s string representation. Note that `SalariedEmployee`’s `getString` calls `Employee`’s protected `getString` to get part of the string representation.

```

1 // Fig. 20.16: 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;
```

Fig. 20.16 | `SalariedEmployee` class derived from `Employee`. (Part I of 2.)

```

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  };

```

Fig. 20.16 | SalariedEmployee class derived from Employee. (Part 2 of 2.)

```

1  // Fig. 20.17: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  #include <stdexcept>
5  #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7  // constructor
8  SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9      : Employee{name} {
10      setSalary(salary);
11 }
12
13 // set salary
14 void SalariedEmployee::setSalary(double salary) {
15     if (salary < 0.0) {
16         throw std::invalid_argument("Weekly salary must be >= 0.0");
17     }
18
19     m_salary = salary;
20 }
21
22 // return salary
23 double SalariedEmployee::getSalary() const {return m_salary;}
24
25 // calculate earnings;
26 // override pure virtual function getPay in Employee
27 double SalariedEmployee::getPay() const {return getSalary();}
28
29 // return a string representation of SalariedEmployee
30 std::string SalariedEmployee::getString() const {
31     return std::format("{}\n{}: ${:.2f}", Employee::getString(),
32                       "salary", getSalary());
33 }

```

Fig. 20.17 | SalariedEmployee class member-function definitions.

Updated Class CommissionEmployee

Our refactored CommissionEmployee class (Figs. 20.18–20.19) has several key changes:

- The class's header (Fig. 20.18) no longer contains prototypes for the earnings and `toString` member functions. These **public non-virtual** base-class functions are now inherited from `Employee`.

- The class’s `private` section now declares overrides for the base class’s `private` pure `virtual` function `getPay` and protected `virtual` function `getString` (Fig. 20.18, lines 24–25). Again, we declared `getString` `private` because this class is `final`, so no other classes can derive from it.
- The class’s member-function implementations (Fig. 20.19) now include overridden function `getPay` (lines 42–44) to return the commission calculation result and overridden function `getString` (lines 47–52) to get a `CommissionEmployee`’s `string` representation. Function `getString` calls `Employee`’s protected `getString` to get part of the `string` representation.

```

1 // Fig. 20.18: 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 };

```

Fig. 20.18 | CommissionEmployee class derived from Employee.

```

1 // Fig. 20.19: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <format>
4 #include <stdexcept>
5 #include "CommissionEmployee.h" // CommissionEmployee class definition
6
7 // constructor
8 CommissionEmployee::CommissionEmployee(std::string_view name,
9                                         double grossSales, double commissionRate) : Employee{name} {
10    setGrossSales(grossSales);

```

Fig. 20.19 | CommissionEmployee class member-function definitions. (Part 1 of 2.)

```

11     setCommissionRate(commissionRate);
12 }
13
14 // set gross sales amount
15 void CommissionEmployee::setGrossSales(double grossSales) {
16     if (grossSales < 0.0) {
17         throw std::invalid_argument("Gross sales must be >= 0.0");
18     }
19
20     m_grossSales = grossSales;
21 }
22
23 // return gross sales amount
24 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
25
26 // set commission rate
27 void CommissionEmployee::setCommissionRate(double commissionRate) {
28     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
29         throw std::invalid_argument(
30             "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 std::string CommissionEmployee::getString() const {
48     return std::format(
49         "{}\n{}: ${:.2f}\n{}: {:.2f}",
50         Employee::getString(),
51         "gross sales", getGrossSales(),
52         "commission rate", getCommissionRate());
53 }
```

Fig. 20.19 | CommissionEmployee class member-function definitions. (Part 2 of 2.)

Runtime Polymorphism with the Employee Hierarchy Using NVI

The test application for this example is identical to the one in Fig. 10.17, so we show only the output in Fig. 20.20. 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 provided 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.

```
EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES
name: Pierre Simon
salary: $800.00
earned $800.00

name: Sierra Dembo
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: Pierre Simon
salary: $800.00
earned $800.00

name: Sierra Dembo
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: Pierre Simon
salary: $800.00
earned $800.00

name: Sierra Dembo
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Fig. 20.20 | Processing Employee derived-class objects with static binding, then polymorphically using dynamic binding.

20.6 Inheriting Base-Class Constructors

A derived class can inherit its base class's constructors. To do so, include a `using` declaration of the following form anywhere in the derived-class definition:

```
using BaseClass::BaseClass;
```

where `BaseClass` is the base class's name. We suggest placing this `using` declaration where you'd typically place the constructor's prototype(s). When the compiler encounters this declaration, it generates a derived-class constructor for each corresponding base-class constructor, passing the derived-class constructor's arguments to the corresponding base-class constructor.

When you inherit constructors:^{22,23}

- Each generated constructor has the same access specifier as its corresponding base-class constructor.
- If a constructor is deleted in the base class by placing = **delete** in its prototype, the corresponding derived-class constructor also is deleted.
- If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.
- If a constructor you explicitly define in the derived class has the same parameter list as a base-class constructor, the derived-class version is used.
- A base-class constructor's default arguments are not inherited. Instead, the compiler generates overloaded constructors in the derived class. For example, if the base class declares the constructor

```
BaseClass(int x = 0, double y = 0.0);
```

the compiler generates the following derived-class constructors without default arguments

```
DerivedClass();  
DerivedClass(int x);  
DerivedClass(int x, double y);
```

These each call the *BaseClass* constructor that specifies the default arguments.

20.7 Multiple Inheritance

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 multiple-inheritance problems are so subtle that newer programming languages, such as Java and C#, support only single inheritance.²⁴ 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.



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 compile. The [isocpp.org FAQ recommends doing multiple inheritance from only pure abstract base classes](https://isocpp.org/faq) to avoid this and other problems we'll discuss here and in Section 20.7.1.²⁵

22. "Using-declaration." Accessed March 10, 2022. https://en.cppreference.com/w/cpp/language/using_declaration

23. C++ Standard, "9.9 The using declaration." Accessed March 10, 2022. <https://timsong-cpp.github.io/cppwp/n4861/namespace.udecl>.

24. More precisely, Java and C# support only single *implementation* inheritance. They do allow multiple interface inheritance.

25. "Inheritance—Multiple and Virtual Inheritance." Accessed April 18, 2023. <https://isocpp.org/wiki/faq/multiple-inheritance>.

Multiple-Inheritance Example

Let's consider a multiple-inheritance example using implementation inheritance (Figs. 20.21–20.25) to reveal a not-so-subtle problem. Class `Base1` (Fig. 20.21) 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`.

```

1 // Fig. 20.21: 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 };

```

Fig. 20.21 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 20.22) is similar to class `Base1`, except its `private` data is a `char` named `m_letter` (line 11). Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns `m_letter`'s value.

```

1 // Fig. 20.22: 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 };

```

Fig. 20.22 | Demonstrating multiple inheritance—`Base2.h`.

Class `Derived` (Figs. 20.23–20.24) inherits from classes `Base1` and `Base2` via multiple inheritance. Class `Derived` has

- a `private` data member of type `double` named `m_real` (Fig. 20.23, line 16),
- a constructor to initialize all the data of class `Derived` (line 12),
- a `public` member function `getReal` that returns the value of `m_real` (line 13) and
- a `public` member function `toString` that returns a `string` representation of a `Derived` object (line 14).

```

1 // Fig. 20.23: Derived.h
2 // Definition of class Derived which inherits
3 // multiple base classes (Base1 and Base2).
4 #pragma once
5 #include <string>
6 #include "Base1.h"
7 #include "Base2.h"
8
9 // class Derived definition
10 class Derived : public Base1, public Base2 {
11 public:
12     Derived(int value, char letter, double real);
13     double getReal() const;
14     std::string toString() const;
15 private:
16     double m_real; // derived class's private data
17 };

```

Fig. 20.23 | Demonstrating multiple inheritance—Derived.h.

```

1 // Fig. 20.24: Derived.cpp
2 // Member-function definitions for class Derived
3 #include <format> // 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 std::string Derived::toString() const {
15     return std::format("int: {}; char: {}; double: {}",
16         Base1::getData(), Base2::getData(), getReal());
17 }

```

Fig. 20.24 | Demonstrating multiple inheritance—Derived.cpp.

For multiple inheritance (in Fig. 20.23), we follow the colon (:) after `class Derived` with a **comma-separated list of base classes** (line 10). In Fig. 20.24, notice that `Derived`'s constructor explicitly calls each base class's constructor using the member-initializer syntax (line 8). The **base-class constructors are called in the order 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 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 20.25 tests the classes in Figs. 20.21–20.24. Line 10 creates `Base1` object `base1` and initializes it to the `int` value 10. Line 11 creates `Base2` object `base2` and initializes it to the `char` value 'Z'. Line 12 creates `Derived` object `derived` and initializes it to contain the `int` value 7, the `char` value 'A' and the `double` value 3.5.

```

1 // fig20_25.cpp
2 // Driver for multiple-inheritance example.
3 #include <iostream>
4 #include <format>
5 #include "Base1.h"
6 #include "Base2.h"
7 #include "Derived.h"
8
9 int main() {
10     Base1 base1{10}; // create Base1 object
11     Base2 base2{'Z'}; // create Base2 object
12     Derived derived{7, 'A', 3.5}; // create Derived object
13
14     // print data in each object
15     std::cout << std::format("{}: {}\n{}: {}\n{}: {}\n\n",
16                             "Object base1 contains", base1.getData(),
17                             "Object base2 contains the character", base2.getData(),
18                             "Object derived contains", derived.toString());
19
20     // print data members of derived-class object
21     // scope resolution operator resolves getData ambiguity
22     std::cout << std::format("{}\n{}: {}\n{}: {}\n{}: {}\n\n",
23                             "Data members of Derived can be accessed individually:",
24                             "int", derived.Base1::getData(),
25                             "char", derived.Base2::getData(),
26                             "double", derived.getReal());
27
28     std::cout << "Derived can be treated as an object"
29             << " of either base class:\n";
30
31     // treat Derived as a Base1 object
32     Base1* base1Ptr{&derived};
33     std::cout << std::format("base1Ptr->getData() yields {}\n",
34                             base1Ptr->getData());
35
36     // treat Derived as a Base2 object
37     Base2* base2Ptr{&derived};
38     std::cout << std::format("base2Ptr->getData() yields {}\n",
39                             base2Ptr->getData());
40 }
```

Fig. 20.25 | Demonstrating multiple inheritance. (Part I of 2.)

```
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. 20.25 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 15–18 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 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 18 gets `derived`'s contents by calling its `toString` member function.

Lines 22–26 output `derived`'s contents again 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()
```

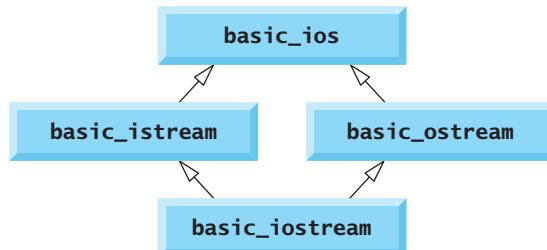
gets the value of `m_letter` inherited from class `Base2`.

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 initializes the `Base1` pointer `base1Ptr` with `&derived`. 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 initializes the `Base2` pointer `base2Ptr` with `&derived`. 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`.

20.7.1 Diamond Inheritance

The preceding example showed 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 the 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 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 Error Produced When Ambiguity Arises in Diamond Inheritance

Figure 20.26 demonstrates the ambiguity that can occur in diamond inheritance. Class `Base` (lines 8–11) contains the 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 sub-object**—the members of class `Base` in this example.

```

1 // fig20_26.cpp
2 // Attempting to polymorphically call a function that is
3 // inherited from each of two base classes.
4 #include <array>
5 #include <iostream>
6
7 // class Base definition
8 class Base {
9 public:
10     virtual void print() const = 0; // pure virtual
11 };
12
  
```

Fig. 20.26 | Attempting to polymorphically call a function that is inherited from each of two base classes. (Part 1 of 2.)

```

13 // class DerivedOne definition
14 class DerivedOne : public Base {
15 public:
16     // override print function
17     void print() const override {std::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 {std::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     std::array<Base*, 3> objects{}; // create array of base-class pointers
39
40     objects[0] = &both; // ERROR--ambiguous
41     objects[1] = &one;
42     objects[2] = &two;
43
44     // polymorphically invoke print
45     for (int i{0}; i < 3; ++i) {
46         objects[i] ->print();
47     }
48 }
```

Microsoft Visual C++ compiler error message:

```
fig20_26.cpp(40,22): error C2594: '=': ambiguous conversions from 'Multiple
*' to '_Ty'
```

Fig. 20.26 | 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—in this case, we used 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 element is assigned the address of an object (lines 40–42). An error occurs when the

address of `both`—an object of class `Multiple`—is assigned to `objects[0]`. The object `both` actually contains two `Base` subobjects. The compiler does not know which subobject the pointer `objects[0]` should point to, so it generates a compilation error indicating an ambiguous conversion.

20.7.2 Eliminating Duplicate Subobjects with `virtual` Base-Class Inheritance



The problem of duplicate subobjects is resolved with `virtual` inheritance. Only one subobject will appear in the derived class when a base class is inherited as `virtual`. Figure 20.27 revises the program of Fig. 20.26 to use a `virtual` base class.

```

1 // fig20_27.cpp
2 // Using virtual base classes.
3 #include <array>
4 #include <iostream>
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 {std::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 {std::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     std::array<Base*, 3> objects{}; // create array of base-class pointers
38}

```

Fig. 20.27 | Using `virtual` base classes. (Part I of 2.)

```

39     objects[0] = &both; // allowed now
40     objects[1] = &one;
41     objects[2] = &two;
42
43     // polymorphically invoke function print
44     for (int i = 0; i < 3; ++i) {
45         objects[i]->print();
46     }
47 }
```

DerivedTwo
DerivedOne
DerivedTwo

Fig. 20.27 | 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. 20.26. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `objects[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

Implementing hierarchies with virtual base classes is simpler if default constructors are used for the base classes. Figures 20.26 and 20.27 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 base-class constructor can be called implicitly.²⁶

ANSI

20.8 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 base-class members' accessibility in a derived class. The first column contains the base-class member access specifiers.

26. “Inheritance—Multiple and Virtual Inheritance : What special considerations do I need to know about when I use virtual inheritance?” Accessed April 18, 2023. <https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-abcs>.

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>

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. With **private** inheritance, **public** and **protected** base-class members become **private** derived-class members.

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 define new types. However, two key differences exist between a **class** and a **struct**. 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

```
1 struct Derived : Base {
2     // ...
3 };
```

uses `public` inheritance by default.

20.9 namespaces: A Deeper Look

You learned in Section 16.6.4 that namespaces help you avoid naming collisions in your code. Namespaces are commonly used in Modern C++ and now in C++20 modules. The code example in Fig. 20.28 explores namespaces in more detail.

```
1 // fig20_28.cpp
2 // Demonstrating namespaces.
3 #include <format>
4 #include <iostream>
5
6 int integer1{98}; // global variable
7
8 // create namespace Example
9 namespace Example {
10    // declare two constants and one variable
11    const double pi{3.14159};
12    const double e{2.71828};
13    int integer1{8};
14
15    void printValues(); // prototype
16
17    // nested namespace
18    namespace Inner {
19        // define variable in nested namespace
20        int year{2023};
21    }
22 }
23
24 // create unnamed namespace
25 namespace {
26     double doubleInUnnamed{88.22}; // declare variable
27 }
28
29 int main() {
30    // output value doubleInUnnamed of unnamed namespace
31    std::cout << std::format("doubleInUnnamed = {}\n", doubleInUnnamed);
32
33    // output global variable
34    std::cout << std::format("(global) integer1 = {}\n", integer1);
35
36    // output values of Example namespace
37    std::cout << std::format(
38        "pi = {}\ne = {}\ninteger1 = {}\nyear = {}\n", Example::pi,
39        Example::e, Example::integer1, Example::Inner::year);
```

Fig. 20.28 | Demonstrating the use of namespaces. (Part I of 2.)

```

40
41     Example::printValues(); // invoke printValues function
42 }
43
44 // display variable and constant values
45 void Example::printValues() {
46     std::cout << "\nIn printValues:\n";
47     std::cout << std::format(
48         "integer1 = {}\npi = {}\ne = {}\n", pi, e, integer1);
49     std::cout << std::format(
50         "doubleInUnnamed = {}\n(global) integer1 = {}\nyear = {}\n",
51         doubleInUnnamed, ::integer1, Inner::year);
52 }

```

```

doubleInUnnamed = 88.22
(global) integer1 = 98
pi = 3.14159
e = 2.71828
integer1 = 8
year = 2023

In printValues:
integer1 = 8
pi = 3.14159
e = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
year = 2023

```

Fig. 20.28 | Demonstrating the use of namespaces. (Part 2 of 2.)

20.9.1 Defining namespaces

Lines 9–22 use the keyword `namespace` to define namespace `Example`, which like a class, has a body delimited by braces (`{}`). `Example`'s members consist of two constants (`pi` and `e` in lines 11–12), an `int` (`integer1` in line 13), a function (`printValues` in line 15) and a **nested namespace** (`Inner` in lines 18–21). `Example`'s `integer1` member has the same name as the global variable `integer1` (line 6). Variables with the same name must have different scopes; otherwise, compilation errors occur.



A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces. Unlike classes, different namespace members can be defined in separate namespace blocks. For example, each standard library header has a namespace block placing its contents in namespace `std`.

Lines 25–27 create an **unnamed namespace** containing `doubleInUnnamed`. Variables, classes and functions in an unnamed namespace are accessible only in the current translation unit. However, unlike variables, classes or functions with `static` linkage, those in the unnamed namespace may be used as template arguments. The unnamed namespace has an implicit `using` directive, so its members appear to occupy the **global namespace**. They are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file. Each separate translation unit has its own unique unnamed namespace.

20.9.2 Accessing namespace Members with Qualified Names

Line 31 outputs the value of `doubleInUnnamed`, which is directly accessible as part of the unnamed namespace. Line 34 outputs global variable `integer1`'s value. For both variables, the compiler first attempts to find a local declaration of the variables in `main`. There are no local declarations, so the compiler assumes those variables are in the global namespace.

Lines 37–39 output the values of `pi`, `e`, `integer1` and `year` from the `Example` namespace. Each must be qualified with `Example::` because the program does not provide any `using` directive or `using` declarations indicating that it will use `Example`'s members. In addition, member `integer1` must be qualified because a global variable has the same name. Otherwise, the global variable's value is output. `year` is a member of the nested namespace `Inner`, so it must be qualified with `Example::Inner::`.

Function `printValues` (defined in lines 45–52) is in the `Example` namespace, so it can access other `Example` members directly without using a namespace qualifier. Lines 47–51 output `integer1`, `pi`, `e`, `doubleInUnnamed`, global variable `integer1` and `year`. Notice that `pi` and `e` are not qualified with `Example`. Variable `doubleInUnnamed` is still accessible because it's in the unnamed namespace, and the variable name does not conflict with any other `Example` members. The global version of `integer1` must be qualified with the scope resolution operator (`::`) because its name conflicts with an `Example` member. Also, `year` must be qualified with `Inner::`. When accessing members of a nested namespace, the members must be qualified with the namespace name unless the member is used within the nested namespace. Placing `main` in a namespace is a compilation error.



20.9.3 using Directives Should Not Be Placed in Headers

namespaces are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. There should never be a `using` directive in a header. This brings the corresponding names into any file that includes the header, which could result in name collisions and subtle, hard-to-find errors. For this reason, you should use only fully qualified names in headers (such as `std::string`).

20.9.4 Nested Namespaces

A nested namespace, such as `Inner` in lines 18–21, may be defined separately from its enclosing namespace. For example, we could remove lines 18–21 and define the nested namespace `Inner` as follows:

```
namespace Example::Inner {
    // define variable in nested namespace
    int year{2003};
}
```

The notation `Example::Inner` indicates that `Example` is the enclosing namespace and `Inner` is a nested namespace of `Example`.

20.9.5 Aliases for namespace Names

namespaces can be aliased.²⁷ This might be useful when dealing with long namespace identifiers or nested namespaces. For example, assuming we have the namespace identifier `CPlusPlusHowToProgram11`, the statement

```
namespace CPPHTTP11 = CPlusPlusHowToProgram11;
```

creates the shorter `namespace alias CPPHTTP11` for `CPlusPlusHowToProgram11`. Similarly, you could define an alias for the nested namespace `Inner` in Fig. 20.28 as follows:

```
namespace Innermost = Example::Inner;
```

20.10 Storage Classes and Storage Duration

As you know, programs use identifiers for variable names, functions and class names. The attributes of variables include `name`, `type`, `size` and `value`. Each identifier in a program also has other attributes, including `scope`, `linkage` and `storage duration`.

As discussed in Section 5.17, an identifier's scope is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. An identifier's linkage determines whether it's known only in the source file where it's declared or across multiple files that are compiled, then linked together. C++20 modules (Chapter 16) introduce `module linkage` for identifiers known only in the module that defines them. An identifier's storage-class specifier helps determine its storage duration and linkage.

Storage Class Specifiers

C++ provides several `storage-class specifiers` that determine a variable's storage duration: `extern`, `mutable`, `static` and `thread_local`. Storage-class specifier `mutable` is used exclusively with classes, and `thread_local` is used in multithreaded applications.

20.10.1 Storage Duration

An identifier's `storage duration` determines the period during which that identifier's storage exists in memory.²⁸ Some exist briefly, some are repeatedly created and destroyed, and others exist for a program's entire execution.

The storage-class specifiers can be split into four storage durations: `automatic`, `static`, `dynamic` and `thread`. In Chapter 11, you learned that you can dynamically allocate additional memory at execution time. Variables allocated dynamically have `dynamic storage duration`. The rest of this section focuses on automatic and static storage duration, and `mutable` data members.

20.10.2 Local Variables and Automatic Storage Duration

Variables with `automatic storage duration` include:

- local variables declared in functions
- function parameters

Such automatic variables are created when program execution enters the block in which they're defined. They exist while the block is active, and they're destroyed when the program exits the block. An automatic variable exists only from where it's defined to the

27. "Namespace aliases." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/namespace_alias.

28. "Storage class specifiers." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/storage_duration.

block's closing brace or for the entire function body in the case of a function parameter. Local variables are of automatic storage duration by default. Automatic storage conserves memory because automatic storage duration variables exist in memory only when the block in which they're defined is executing.



20.10.3 Static Storage Duration

Keywords `extern` and `static` declare identifiers for variables with **static storage duration** and functions. Variables with static storage duration exist in memory from the point at which the program begins execution until the program terminates. Such a variable is initialized once when its declaration is encountered. A function name exists when the program begins execution. Even though function names and static-storage-duration variables exist from the start of program execution, their scope determines where they can be used in the program.

Identifiers with Static Storage Duration

There are two types of identifiers with static storage duration—external identifiers (such as global variables) and local variables declared with the storage-class specifier `static`. **Global variables** are created by placing variable declarations outside any class or function definition. Global variables retain their values throughout a program's execution. Global variables and global functions can be referenced by any function following their declarations or definitions in the source file. If a global variable is declared `static`, it is known only in that translation unit. If a C++20 module (Chapter 16) contains a global `static` variable, the module may not export that variable. You should avoid global variables, where possible.

static Local Variables

Local `static` variables are known only in the function that declares them but retain their values when the function returns to its caller. The next time the function is called, each `static` local variable contains the value it had when the function last completed execution. The following statement declares the local `static` variable `count` and initializes it to 1:

```
static int count{1};
```

The initialization occurs the first time this statement is encountered. All numeric variables of static storage duration are initialized to zero by default. It's nevertheless a good practice to explicitly initialize every variable so its initial value is clear.

Storage-class specifiers `extern` and `static` determine an identifier's **linkage** when they're applied to external identifiers such as global variables and global function names:²⁹

- `extern` indicates that the identifier is visible to other translation units.
- `static` indicates that the identifier is visible only in the current translation unit.

C++20 modules (Chapter 16) must explicitly `export` identifiers to make them visible to other translation units.

29. "Storage class specifiers." Accessed April 18, 2023. https://en.cppreference.com/w/cpp/language/storage_duration.

20.10.4 **mutable** Class Members

Section 15.9.2 introduced the `const_cast` operator for removing the “`const`-ness” of a type. This can be applied to a data member of a `const` object from the body of a `const` member function of that object’s class. This enables the `const` member function to modify the data member, even though the member function considers the object to be `const`.

For example, consider a linked list that maintains its contents in sorted order. Searching the list does not require modifying its data, so the search function could be a `const` member function. However, it’s conceivable that a linked-list object—to make subsequent searches more efficient—might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.



For a class with a “secret” implementation detail that should always be modifiable—such as our linked-list example—the C++ Core Guidelines recommend using the storage-class specifier `mutable` to designate that a data member is always modifiable, even in a `const` member function or `const` object.^{30,31} Though `mutable` is a storage-class specifier, it does not affect a variable’s storage duration or linkage.

Figure 20.29 presents a mechanical example of a `mutable` member to prove that such a member is always modifiable. Class `TestMutable` (lines 6–15) contains

- a constructor (line 8),
- function `getValue` (lines 10–12), which is a `const` member function that returns a copy of `value`, and
- a `private` data member `value` that’s declared `mutable` (line 14).

Function `getValue` increments the `mutable` data member `value` in the `return` statement (line 11). Typically, a `const` member function cannot modify the object on which it’s called. Because the data member `value` is `mutable`, this `const` function can modify the data.

```

1 // fig20_29.cpp
2 // Demonstrating storage-class specifier mutable.
3 #include <iostream>
4
5 // class TestMutable definition
6 class TestMutable {
7 public:
8     TestMutable(int v = 0) : value{v} { }
9
10    int getValue() const {
11        return ++value; // increments value
12    }

```

Fig. 20.29 | Demonstrating a `mutable` data member. (Part 1 of 2.)

30. C++ Core Guidelines, “ES.50: Don’t cast away `const`.” Accessed April 18, 2023. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>.

31. “cv (`const` and `volatile`) type qualifiers—`mutable` specifier.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/cv>.

```

13  private:
14      mutable int value; // mutable member
15  };
16
17 int main() {
18     const TestMutable test{99};
19
20     std::cout << "Initial value: " << test.getValue()
21     << "\nModified value: " << test.getValue() << "\n";
22 }

```

```

Initial value: 100
Modified value: 101

```

Fig. 20.29 | Demonstrating a `mutable` data member. (Part 2 of 2.)

Line 18 declares the `const TestMutable` object `test` and initializes it to 99. Line 20 calls the `const` member function `getValue`, which adds one to `value` and returns its new value. The compiler allows the `getValue` call on the object `test` because it's a `const` object, and `getValue` is a `const` member function. However, `getValue` modifies `value`. Thus, when line 20 invokes `getValue`, the new value (100) is output to prove that the `mutable` data member was modified. Line 21 demonstrates this again, displaying 101.

20.11 Operator Keywords

The following table shows the **operator keywords** you can use instead of corresponding C++ operators.

Operator	Operator keyword	Description
<i>Logical operator keywords</i>		
<code>&&</code>	<code>and</code>	logical AND
<code> </code>	<code>or</code>	logical OR
<code>!</code>	<code>not</code>	logical NOT
<i>Inequality operator keyword</i>		
<code>!=</code>	<code>not_eq</code>	inequality
<i>Bitwise operator keywords</i>		
<code>&</code>	<code>bitand</code>	bitwise AND
<code> </code>	<code>bitor</code>	bitwise inclusive OR
<code>^</code>	<code>xor</code>	bitwise exclusive OR
<code>~</code>	<code>compl</code>	bitwise complement
<i>Bitwise assignment operator keywords</i>		
<code>&=</code>	<code>and_eq</code>	bitwise AND assignment
<code> =</code>	<code>or_eq</code>	bitwise inclusive OR assignment
<code>^=</code>	<code>xor_eq</code>	bitwise exclusive OR assignment

Figure 20.30 demonstrates several operator keywords. The program declares and initializes two `bool` variables (lines 7–8). Lines 13–19 use the logical operator keywords to perform various logical operations with `bool` variables `a` and `b`.

```

1 // fig20_30.cpp
2 // Demonstrating operator keywords.
3 #include <format>
4 #include <iostream>
5
6 int main() {
7     bool a{true};
8     bool b{false};
9
10    std::cout << std::format("a = {}; b = {}\n\n", a, b);
11
12    std::cout << "Logical operator keywords:\n"
13        << std::format("  a and a: {}", a and a)
14        << std::format("  a and b: {}", a and b)
15        << std::format("  a or a: {}", a or a)
16        << std::format("  a or b: {}", a or b)
17        << std::format("  not a: {}", not a)
18        << std::format("  not b: {}", not b)
19        << std::format("a not_eq b: {}", a not_eq b);
20 }
```

```

a = true; b = false
Logical operator keywords:
  a and a: true
  a and b: false
  a or a: true
  a or b: true
  not a: false
  not b: true
a not_eq b: true
```

Fig. 20.30 | Demonstrating operator keywords.

20.12 decltype Operator

The `decltype` operator determines an expression's type at compile time.³² If the expression is a function call, `decltype` determines the function's return type, which for a function template often changes based on the template's type argument(s).

The `decltype` operator is particularly useful in complex template metaprogramming where it can be challenging to provide or even determine a proper type declaration. Rather than writing a complex type declaration, you can place an expression in the parentheses of `decltype` and let the compiler “figure it out.” The C++ standard library class templates do this frequently.

32. “`decltype` specifier.” Accessed April 18, 2023. <https://en.cppreference.com/w/cpp/language/decltype>.

The format of a `decltype` expression is

```
decltype(expression)
```

The expression is not evaluated. `decltype` is commonly used with trailing return types (Section 20.13). You also can use `decltype(auto)` to infer a function's return type as in:

```
decltype(auto) functionName(parameters)
```

For more information on `decltype`, see

<https://timsong-cpp.github.io/cppwp/n4861/dcl.type decltype>

20.13 Trailing Return Types for Functions

Section 6.14.2 introduced lambda expressions. As you saw, lambdas that return values are declared with trailing return types in which the return type is specified to the right of the parameter list. This syntax also can be used with functions and member functions. To specify a trailing return type, place the keyword `auto` before the function name, then follow the function's parameter list with `->` and the return type. For example, to specify a trailing return type for a function template `maximum` that finds the largest of three values of the same type, `T`, you'd write

```
template <typename T>
auto maximum(T x, T y, T z) -> T {
    // statements
}
```

There are cases for which only trailing return types are appropriate, such as when the return type depends on the value of an expression using the function's parameters. For example, the following function template multiplies two values of possibly different types:

```
template <typename T, typename U>
auto multiply(T left, U right) -> decltype(left * right) {
    return left * right;
}
```

When instantiating the `multiply` template, the compiler looks at the argument types passed to `left` and `right`. It uses those types to determine the type that would be produced by the expression in `decltype`'s parentheses. Consider the call

```
multiply(10, 7.3)
```

which receives an `int` and a `double`. Recall that in a mixed-type expression, an `int` will be promoted to a `double`, so multiplying 10 by 7.3 produces a `double`. The trailing return type with `decltype` enables the compiler to customize `multiply`'s return type at compile time based on the function's arguments. For more information on trailing return types, see

<https://timsong-cpp.github.io/cppwp/n4861/dcl.fct>

20.14 [[nodiscard]] Attribute

Some functions return values that you should not ignore. For example, Section 2.8 introduced the `string` member function `empty`. When you want to know whether a `string` is empty, you must not only call `empty` but also check its return value in a condition, such as:

```

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

```

In C++20, `string`'s `empty` function is declared with the `[[nodiscard]]` attribute—the compiler issues a warning if the caller does not use the return value.³³ Since C++17, many library functions have been enhanced with `[[nodiscard]]` to help you write correct code.



`[[nodiscard("with reason")]]` Attribute

You can include a descriptive message that the compiler will display as part of the warning message, as in:

```
[[nodiscard("Insight goes here")]]
```

You may also use this attribute on your own function definitions. Figure 20.31 shows a `cube` function declared with `[[nodiscard]]`. You place the attribute before the return type—typically on a line by itself for readability (line 4). Line 10 calls `cube` but does not use the returned value. The three outputs show the compiler warnings from Visual C++, `clang++` and `g++`, respectively. These are just warnings, so the program still compiles and runs. The primary purpose of this example is to demonstrate the compiler warning messages produced by `[[nodiscard]]`. This program does not produce any output when it runs because it ignores `cube`'s return value when it should not.

```

1 // fig20_31.cpp
2 // [[nodiscard]] attribute.
3
4 [[nodiscard("Do not ignore! Otherwise, you won't know the cube of x")]]
5 int cube(int x) {
6     return x * x * x;
7 }
8
9 int main() {
10     cube(10); // generates a compiler warning
11 }
```

C:\Users\pauldeitel\examples\ch20\fig20_31.cpp(10,8): warning C4858: discarding return value: Do not ignore! Otherwise, you won't know the cube of x.

fig20_31.cpp:10:4: warning: ignoring return value of function declared with
'nodiscard' attribute: Do not ignore! Otherwise, you won't know the cube of
x. [-Wunused-result]
cube(10); // generates a compiler warning
^~~~ ~~
1 warning generated.

Fig. 20.31 | `[[nodiscard]]` attribute.

33. “9.12.8 Nodiscard attribute.” Accessed April 18, 2023. <https://timsong-cpp.github.io/cppwp/n4861/dcl.attr.nodiscard>.

```
fig20_31.cpp: In function 'int main()':  
fig20_31.cpp:10:8: warning: ignoring return value of 'int cube(int)',  
declared with attribute 'nodiscard': 'Do not ignore! Otherwise, you won't  
know the cube of x.' [-Wunused-result]  
10 |     cube(10); // generates a compiler warning  
   | ~~~~~^~~~~~  
fig20_31.cpp:5:5: note: declared here  
 5 | int cube(int x) {  
   | ^~~~
```

Fig. 20.31 | `[[nodiscard]]` attribute.

20.15 Some Key C++23 Features

As we completed this book, the C++23 standard was close to completion, and there was little compiler support, so this section's code snippets and sample outputs are made up and have not been tested. Here, we overview several key C++23 features.^{34,35,36} Previously, we've mentioned a few C++23 features and some possible future C++ features including

- the `std::mdarray` container (Section 6.13),
- contracts (Section 12.13),
- ranges enhancements (Section 14.10),
- the modularized standard library (Section 16.12),
- concurrent data structures (Section 17.15),
- parallel ranges algorithms (Section 17.15) and
- executors (Sections 18.5 and 18.8).

The `cppreference.com` table showing compiler support for C++23

https://en.cppreference.com/w/cpp/compiler_support/23

lists over 120 items, with some encompassing many new features. Many of these are summarized at

<https://en.wikipedia.org/wiki/C%2B%2B23>

C++23 is a different kind of update from C++20, which had four major new features—ranges, concepts, modules and coroutines.

There are 25 new C++23 items categorized as “defect report” improvements (labeled as DR in the compiler-support table)—these are a combination of bug fixes for existing features. Other items are new features or enhancements to existing features, such as new ranges and views capabilities that provide additional functional-style programming capabilities.

34. “C++23.” Wikipedia. Wikimedia Foundation. Accessed April 18, 2023. <https://en.wikipedia.org/wiki/C%2B%2B23>.

35. “Compiler support for C++23.” Accessed April 18, 2023. https://en.cppreference.com/w/cpp/compiler_support/23.

36. Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, Billy Baker, Nevin Liber, Ville Voutilainen and Inbal Levi, “Library Evolution Plan for Completing C++23,” November 9, 2021. Accessed April 18, 2023. <https://wg21.link/p2489>.

C++23 Formatted Output

We used C++20's `std::format` function for text formatting throughout the book, and Chapter 19 presented many more C++20 text formatting features. C++23's `<print>` header provides the functions `std::print` and `std::println` ("print line"), which output formatted strings directly to the standard output stream by default. They also provide overloads for writing formatted text to other streams. `std::println` automatically outputs a newline character after outputting its arguments.

Assume the `int` variable `total` exists and contains 385. With the new formatted output functions, a statement like

```
std::cout << std::format("Sum is {}\n", total);
```

can be written more concisely with `std::print` as

```
std::print("Sum is {}\n", total);
```

or with `std::println` as

```
std::println("Sum is {}", total);
```

Each of the preceding statements would produce the following output and move the output cursor to the beginning of the next line:

```
Sum is 385
```

For more information on `std::print` and `std::println`, see

<https://en.cppreference.com/w/cpp/header/print>

C++23 Ranges Enhancements

C++20's ranges and views have been enhanced with 11 new `std::ranges` algorithms and 14 new `std::views` functional-style programming capabilities. Some key new features include:

- `ranges::to` enables you to convert a range into a container of elements, making it convenient to store the results of each functional-style pipeline of operations. For example, given the lambda expression `isEven` (line 1), which returns `true` if its argument is divisible by two, the pipeline in lines 3–6) produces integers starting from 0 (line 3), keeps only the integers for which `isEven` returns `true` (line 4), takes the first five of those even integers (line 5) and collects the pipeline's results into a `vector` (line 6). After this code executes, the variable `results` is a `vector<int>` containing 0, 2, 4, 6 and 8.

```

1 auto isEven=[](int x) {return x % 2 == 0;};
2 auto results{
3     std::views::iota(0)
4     | std::views::filter(isEven)
5     | std::views::take(5)
6     | std::ranges::to<std::vector>()
```

- Algorithms for searching for items in ranges, such as `ranges::contains`, `ranges::contains_subrange`, `ranges::find_last`, `ranges::find_last_if`, `ranges::find_last_if_not`, `ranges::starts_with` and `ranges::ends_with`.
- View operations `views::zip` and `views::zip_transform` for processing parallel ranges of elements. For example, the following snippet uses `views::zip` to produce tuples containing corresponding elements from the `studentNames` and `gradePointAverages` vectors, then display the tuples' elements:

```

1 std::vector studentNames{"Meriem", "Pierre", "Sierra"};
2 std::vector gradePointAverages{3.9, 3.5, 4.0};
3
4 for (std::tuple<std::string&, float&> student :
5     std::views::zip(studentNames, gradePointAverages)) {
6     std::println("{}: {}", 
7         std::get<0>(student), std::get<1>(student));
8 }
```

Meriem: 3.9
Pierre: 3.5
Sierra: 4.0

- View operation `views::enumerate` provides a convenient mechanism for accessing an element's index and value in a range. For example, the following code iterates through the vector `colors`, displaying each element's index and value without the need for a mutable counter variable:

```

1 std::vector colors{"red", "orange", "yellow"};
2
3 for (const auto& [i, color] : std::views::enumerate(colors)) {
4     std::println("{}: {}", i, color);
5 }
```

0: red
1: orange
2: yellow

For more information on C++23 ranges and views enhancements, see the items marked C++23 at

<https://en.cppreference.com/w/cpp/header/ranges>

C++23 Generator Coroutines

In Section 18.4, we created a generator coroutine using Sy Brand's `t1::generator` library. C++23 now provides `std::generator`, so you can create generator coroutines without a third-party library. The `std::generator` should be a one-for-one replacement for `t1::generator` in Fig. 18.1. For more info on `std::generator` see:³⁷

37. The link beginning with <https://wg21.link> take you to the most up-to-date versions of the C++ standard committee papers that describe new features.

<https://wg21.link/p2502>
<https://en.cppreference.com/w/cpp/header/generator>

C++23 Multidimensional Subscript Operator

Section 6.13 introduced multidimensional arrays and showed that one way to access each element is by using separate [] for each dimension—given a two-dimensional array of ints named `values`, you can access each element with an expression of the form:

```
values[row][column]
```

Unfortunately, the preceding syntax does not work with custom classes, but programmers can overload the function-call operator () , which can take an arbitrary number of parameters. In a class representing two-dimensional data, you could use an expression like the following to invoke the overloaded () operator on an object called `values`:

```
values(row, column)
```

C++23 enables custom types to overload the [] operator for multiple dimensions by allowing an arbitrary number of comma-separated parameters, like the overloaded () operator. So the preceding expression may be implemented with the notation

```
values[row, column]
```

C++23 std:::mdspan

Section 7.10 introduced `std::span` for creating views of contiguous elements in containers such as arrays, vectors and built-in arrays. C++23 provides `std:::mdspan` for creating **multidimensional views of contiguous data**. It also supports selecting **subviews** of that data, known as **slices**. Class `mdspan` implements an overloaded multidimensional [] operator for accessing elements in an `mdspan`'s view. The following code snippet uses an `mdspan` (line 2) to view a one-dimensional, six-element array of ints as 2-by-3 data and displays the elements by row:

```

1  std::array values{2, 3, 5, 7, 11, 13};
2  auto values2D{std:::mdspan(values.data(), 2, 3)};
3
4  for (size_t row{0}; row != values2D.extent(0); ++row) {
5      for (size_t column{0}; column != values2D.extent(1); ++column) {
6          std::print("{:>2d} ", values2D[row, column]);
7      }
8      std::println("");
9  }
```

2 3 5
7 11 13

For more information on `mdspan`, see:

<https://en.cppreference.com/w/cpp/container/mdspan>

C++23 Container Enhancements

There are several C++23 container enhancements:

- You can now initialize stack and queue container adapters from iterator pairs representing ranges of elements in other containers. For example, the following code snippet initializes a `queue` from a `vector` of `ints`. For more information, see <https://wg21.link/p1425>.

```

1 std::vector values{2, 3, 5, 7, 11, 13};
2 std::queue myQueue{values.begin(), values.end()};

```

- The associative containers (Section 13.11) enable heterogeneous lookup—you can search for keys using values of compatible types without first converting those values to the container’s key type. For example, if the keys are `std::strings`, you can pass search functions C-style strings or `std::string_views`. This heterogeneous lookup capability is now supported for the associative container member functions `erase` and `extract`. For more information, see <https://wg21.link/p2077>.
- The `<flat_map>` and `<flat_set>` headers provide container adaptors for manipulating sorted sequence containers (vectors by default) as maps and sets (Section 13.11). These new container adaptors provide better performance characteristics than the sorted associative containers. For more information, see  https://www.sandordargo.com/blog/2022/10/05/cpp23-flat_map

https://en.cppreference.com/w/cpp/header/flat_map

https://en.cppreference.com/w/cpp/header/flat_set

C++23 Attribute `[[assume]]`

You can indicate assumptions the compiler should make about your source code so it can optimize the resulting object code. For example, line 2 in the following snippet tells the compiler to assume `denominator` is not equal to 0, so the compiler does not need to generate any code for division by 0, which might result in more compact or faster code: 

```

1 int quotient(int numerator, int denominator) {
2     [[assume(denominator != 0)]];
3     return numerator / denominator;
4 }

```

For more information, see

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

C++23 `<stacktrace>` Header

Chapter 12 discussed C++’s exception-handling mechanism. Various programming languages, such as Java, C# and Python, enable programmatic access to information about the functions on the function-call stack when an exception occurs. The new `<stacktrace>` header adds these capabilities to C++. For more information, see

<https://en.cppreference.com/w/cpp/header/stacktrace>

Features Being Considered for Future C++ Versions

- **Asynchronous execution capabilities:** The proposed new features include three kinds of components—`schedulers`, `senders`, and `receivers`—and customizable

asynchronous algorithms. For the current version of this paper, visit <https://wg21.link/P2300>.

- **Passing list initializers to standard algorithms:** In Section 11.6, we demonstrated creating a function that could receive a list initializer as an argument. The paper “P2248: Enabling List-Initialization For Algorithms” proposes new standard library algorithm overloads that accept initializer lists as arguments. For the current version of this paper, visit <https://wg21.link/P2248>.

Videos References

Check out the following videos for more C++23 details:

- Sy Brand, “What’s New In C++23.”
<https://www.youtube.com/watch?v=mQzijnbnT04>
- Timur Doumler, “How C++23 changes the way we write code.”
<https://www.youtube.com/watch?v=OH3PnMe02gs>
- Marc Gregoire, “C++23 — What’s In It For You?”
<https://www.youtube.com/watch?v=b0NkuoUkv0M>

20.16 Wrap-Up

In this chapter, we overviewed additional C++ topics. We showed how to use the smart pointer standard library class templates `shared_ptr` and `weak_ptr` for managing shared resources. We demonstrated how to use custom deleter functions to allow `shared_ptrs` to manage resources that require special destruction procedures. We also explained how `weak_ptrs` can be used to prevent memory leaks in circularly referential data.

Next, we presented another way to implement runtime polymorphism by processing objects of classes not related by inheritance, using duck typing via the `std::variant` class template and the standard-library function `std::visit`. We introduced the `protected` access specifier—derived-class member functions and `friends` of the derived class can access `protected` base-class members.

We introduced the non-virtual interface (NVI) idiom in which each base-class function serves only one purpose—either as 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 showed how to inherit base-class constructors, enabling the compiler to generate derived-class constructors that match the base class ones—this is particularly convenient when derived-class constructors simply call corresponding base-class constructors. We demonstrated multiple inheritance, discussed its potential problems and showed how `virtual` inheritance can solve them. 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.

We discussed storage classes and storage duration, and how they affect object lifetimes. We showed how to use namespaces to avoid naming conflicts. You saw that some operator symbols can be represented as operator keywords.

We discussed trailing return types for functions and showed that the `decltype` operator enables the compiler to determine an expression's type at compile time. Both of these capabilities are used extensively in template metaprogramming (Chapter 15).

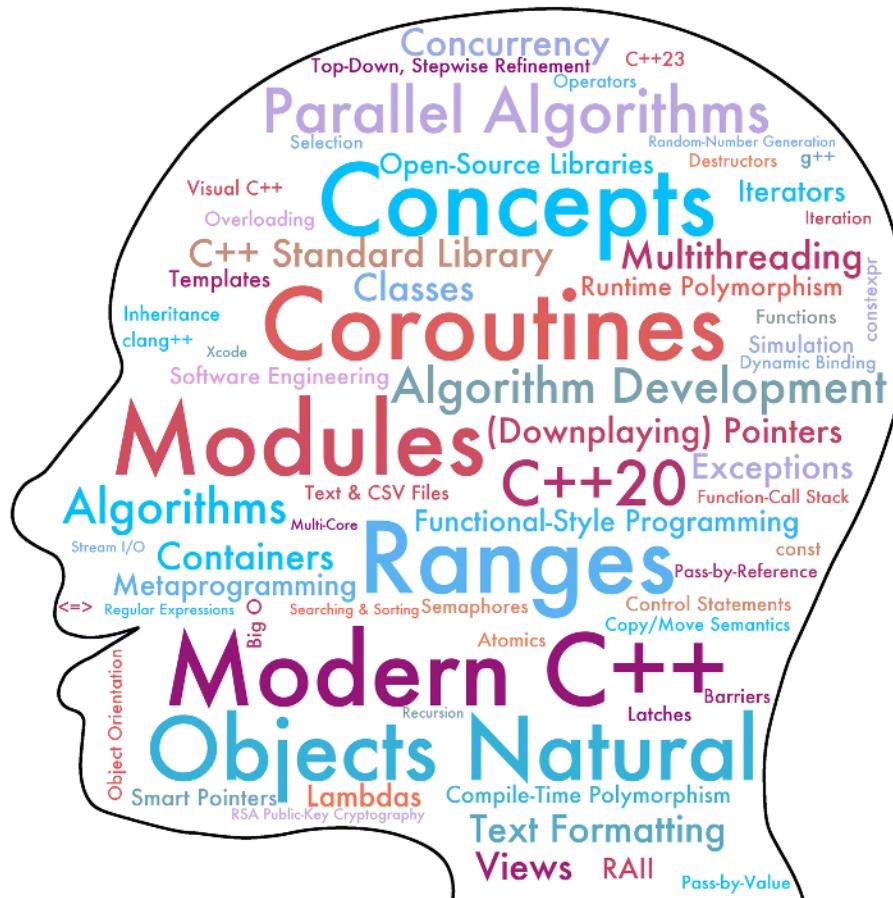
We presented the `[[nodiscard]]` attribute for indicating that a function's return value should not be ignored, and used C++20's `[[nodiscard]]` enhancement to specify why the return value should not be ignored. The compiler displays the reason in a warning message if you do not use a `[[nodiscard]]` function's return value.

Finally, we presented some key new features in the recently released C++23 standard. In the next chapter, we implement various searching and sorting algorithms. We also discuss Big O notation for expressing algorithm efficiency and use it to compare the performance of various popular searching and sorting algorithms.

This page intentionally left blank

21

Computer Science Thinking: Searching, Sorting and Big O



Objectives

In this chapter, you'll:

- Use Big O notation to express the efficiency of searching and sorting algorithms and to compare their performance.
- Search for a given value in an array using linear search and binary search.
- Sort an array using insertion sort, selection sort and the recursive merge sort algorithms.
- Understand the nature of algorithms of constant, linear and quadratic runtime.

Outline

21.1 Introduction	21.6 Selection Sort
21.2 Efficiency of Algorithms: Big O	21.6.1 Implementation
21.2.1 $O(1)$ Algorithms	21.6.2 Efficiency of Selection Sort
21.2.2 $O(n)$ Algorithms	21.7 Merge Sort (A Recursive Implementation)
21.2.3 $O(n^2)$ Algorithms	21.7.1 Implementation
21.3 Linear Search	21.7.2 Efficiency of Merge Sort
21.3.1 Implementation	21.7.3 Summarizing Various Algorithms' Big O Notations
21.3.2 Efficiency of Linear Search	
21.4 Binary Search	21.8 Wrap-Up
21.4.1 Implementation	Exercises
21.4.2 Efficiency of Binary Search	
21.5 Insertion Sort	
21.5.1 Implementation	
21.5.2 Efficiency of Insertion Sort	

21.1 Introduction

In Section 6.12, you learned that sorting places data in ascending or descending order based on one or more sort keys, and searching determines whether an array contains a value that matches a particular key value. In this chapter, we first introduce **Big O notation** to characterize how much work algorithms may need to do to solve problems, so we can compare the efficiency of the searching and sorting algorithms we present. Next, we present two popular search algorithms—the simple **linear search** (Section 21.3) and the faster but more complex **binary search** (Section 21.4). Then, we present three sorting algorithms—**insertion sort** (Section 21.5), **selection sort** (Section 21.6) and the more efficient but more complex recursive **merge sort** (Section 21.7).

When sorting arrays, the result will be the same regardless of the sorting algorithm, so your algorithm choice affects only the program's run time and memory use. As you'll see, algorithms that are easy to program (such as linear search, insertion sort and selection sort) are typically less efficient than algorithms that are harder to program (such as binary search and recursive merge sort).

The exercises present additional sorting algorithms, such as the **recursive quicksort** and the **bucket sort**, which achieves high performance by cleverly using considerably more memory than the other sorts we discuss. The table below summarizes the searching and sorting algorithms presented in this book's examples and exercises:

Algorithm	Location	Algorithm	Location
<i>Searching Algorithms</i>			
Linear search	Section 21.3	Insertion sort	Section 21.5
Binary search	Section 21.4	Selection sort	Section 21.6
Recursive linear search	Exercise 21.8	Recursive merge sort	Section 21.7
Recursive binary search	Exercise 21.9	Bubble sort	Exercises 21.5–21.6
<code>binary_search</code> standard library function	Sections 6.12 and 14.4.6	Bucket sort	Exercise 21.7
		Recursive quicksort	Exercise 21.10
		<code>sort</code> standard library function	Sections 6.12 and 14.4.6

A Note About This Chapter's Examples

The searching and sorting algorithms in this chapter are implemented as function templates that manipulate objects of the array class template. Some examples display array-element values throughout the searching or sorting process to help you visualize how the algorithms work. These outputs slow an algorithm's performance and are not included in  Perf industrial-strength code.



Checkpoint

- 1 *(Fill-in)* _____ characterizes how much work algorithms may need to do to solve problems.

Answer: Big O notation.

- 2 *(Discussion)* What is the key advantage of binary search over linear search?

Answer: Binary search is faster than linear search (but it's more complex to implement).

- 3 *(Discussion)* Why might you prefer insertion sort to merge sort?

Answer: Insertion sort is easier to implement (but it's less efficient).

21.2 Efficiency of Algorithms: Big O



One way to describe an algorithm's 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 mainly on how many data elements there are. In this chapter, we use Big O to describe various sorting algorithms' worst-case run times.

21.2.1 $O(1)$ Algorithms

Suppose an algorithm tests whether an array's first element is equal to its second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1,000 elements, the algorithm still requires one comparison. In fact, the algorithm is completely independent of the array's number of elements. This algorithm is said to have **constant run time**, which we represent in Big O notation as $O(1)$ and pronounce "order 1." An $O(1)$ algorithm does not necessarily require only one comparison. $O(1)$ means that the number of comparisons is *constant*—it does not grow as the array size increases. An algorithm that tests whether the first array element is equal to any of the next three elements is still $O(1)$, even though it requires three comparisons.

21.2.2 $O(n)$ Algorithms

An algorithm that tests whether an array's first element is equal to *any* of the array's other elements requires at most $n - 1$ comparisons, where n is the array's number of elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1,000 elements, this algorithm requires up to 999 comparisons.

As n grows larger, the n in the expression $n - 1$ "dominates," so subtracting 1 becomes inconsequential. Big O is designed to highlight these dominant terms and ignore those that become unimportant as n grows. For this reason, an algorithm that requires $n - 1$ comparisons 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 just "order n ."

21.2.3 $O(n^2)$ Algorithms

Suppose an algorithm tests whether *any* array element is duplicated elsewhere in the array. The algorithm compares the first element with all of the array's other elements. The algorithm then compares the second element with all of the array's other elements except the first—the second was already compared to the first. Then, the algorithm compares the third element with all the other elements except the first two. In the end, this algorithm makes a total of $(n - 1) + (n - 2) + \dots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Big O notation highlights the n^2 term, leaving $n^2/2$. But as we'll soon see, constant factors (such as $1/2$) are omitted in Big O notation.

Big O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, the algorithm will require 64 comparisons. With this algorithm, doubling the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, the algorithm will require 32 comparisons. Again, doubling the number of elements *quadruples* the number of comparisons. Both algorithms grow as the square of n , so Big O ignores the constant and both algorithms are considered to be $O(n^2)$. This is referred to as **quadratic run time** and pronounced “on the order of n -squared” or simply “order n -squared.”

When n is small, $O(n^2)$ algorithms (running on today's billion-operations-per-second personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element array would require a trillion “operations,” where each could require several machine instructions. This could require a few hours to execute. A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! As you'll see in this chapter, $O(n^2)$ algorithms are easy to write. You'll also see an algorithm with a more favorable Big O measure. Efficient algorithms often require clever coding and more work to create. Their superior performance can be worth the extra effort, especially as n gets large.



Checkpoint

- 1** (True/False) $O(n^2)$ algorithms running on today's billion-operations-per-second personal computers will not noticeably affect performance.

Answer: False. Actually, when n is *small*, $O(n^2)$ algorithms will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation, even on today's powerful systems.

- 2** (True/False) An algorithm that is $O(1)$ requires only one comparison.

Answer: False. $O(1)$ means the number of comparisons is *constant*. The algorithm may require multiple comparisons, but that number does not grow as the array's size increases.

- 3** (Fill-In) An $O(n)$ algorithm is referred to as having a _____ run time.

Answer: linear.

21.3 Linear Search

The simple **linear search** can determine whether an unsorted array—that is, an array with element values that are in no particular order—contains a specified search key. Exercise 21.8 asks you to implement a recursive linear search.

21.3.1 Implementation

Function template `linearSearch` (Fig. 21.1, lines 10–19) compares each element of an array with a **search key** (line 13). Because the array is not in any particular order, it's just as likely that the search key will be found in the first element as the last. On average, therefore, the program must compare the search key with **half** of the array's elements. The program must compare the search key to every array element to determine that a value is not in the array. Linear search works well for small or unsorted arrays but is inefficient for large arrays. If the array is **sorted** (that is, its elements are in ascending order), you can use the high-speed binary search technique (Section 21.4).

```
1 // Fig. 21.1: LinearSearch.cpp
2 // Linear search of an array.
3 #include <iostream>
4 #include <array>
5 #include <format>
6
7 // compare key to every element of array until location is
8 // found or until end of array is reached; return location of
9 // element if key is found or -1 if key is not found
10 template <typename T, size_t size>
11 int linearSearch(const std::array<T, size>& items, const T& key) {
12     for (int i{0}; i < items.size(); ++i) {
13         if (key == items[i]) { // if found,
14             return i; // return location of key
15         }
16     }
17     return -1; // key not found
18 }
19
20
21 int main() {
22     std::array<int, 100> arrayToSearch;
23
24     for (int i{0}; i < arrayToSearch.size(); ++i) {
25         arrayToSearch[i] = 2 * i; // create some data
26     }
27
28     std::cout << "Enter integer search key: ";
29     int searchKey; // value to locate
30     std::cin >> searchKey;
31
32     // attempt to locate searchKey in arrayToSearch
33     int element{linearSearch(arrayToSearch, searchKey)};
34 }
```

Fig. 21.1 | Linear search of an array. (Part I of 2.)

```

35  // display results
36  if (element != -1) {
37      std::cout << std::format("Found value in element {}\n", element);
38  }
39  else {
40      std::cout << "Value not found\n";
41  }
42 }
```

Enter integer search key: 36
Found value in element 18

Enter integer search key: 37
Value not found

Fig. 21.1 | Linear search of an array. (Part 2 of 2.)

Perf 21.3.2 Efficiency of Linear Search

The linear search algorithm runs in $O(n)$ time. Linear search's worst case is checking every element to determine whether the search key is in the array. If the array's size **doubles**, so does the number of comparisons. 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. If a program needs to perform many searches on large arrays, it may be better to implement a more efficient algorithm, such as the **binary search** (Section 21.4).

Perf  Sometimes the simplest algorithms perform poorly—their virtue is that they're easy to program, test and debug. More complex algorithms are typically required to maximize performance.



Checkpoint

1 (*True/False*) A linear search is typically used to determine whether a sorted array contains a specified search key.

Answer: False. Actually, linear search is typically used to determine whether an **unsorted** array contains a specified search key.

2 (*Multiple Choice*) Which of the following statements about linear search is *false*?

- It compares every array element with the search key.
- Because the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last.
- On average, it compares the search key with half the array's elements.
- It works well for small or unsorted arrays but it's inefficient for large arrays.

Answer: a) is *false*. It could find a match before reaching the array's end, in which case it would terminate the search before comparing every element with the search key.

21.4 Binary Search

The **binary search algorithm** is more efficient than linear search but requires a **sorted array**. This is only worthwhile when the array, once sorted, will be searched many times—or when the application has stringent performance requirements. The first iteration of this algorithm tests the **middle array element**. If this matches the search key, the algorithm ends. Assume the array is sorted in ascending order. If the search key is **less than the middle element**, the search key cannot match any element in the array's second half. So, the algorithm continues with only the **first half**—the first element up to, but not including, the middle element. If the search key is **greater than the middle element**, the search key cannot match any element in the array's first half. So, the algorithm continues with only the **second half**—the element after the middle element through the last element. Each iteration tests the **middle value of the array's remaining elements**. If the element does not match the search key, the algorithm eliminates half the remaining elements. The algorithm ends by

- finding an element that matches the search key or
- reducing the sub-array size to zero, indicating the search key was not found.



Binary Search of 15 Integer Values

As an example, consider the sorted 15-element array

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

Let's use the binary search algorithm to find the search key 65. A binary search first checks whether the middle element (51) is the search key. The search key (65) is greater than 51, so 51 is eliminated from consideration along with the array's first half—that is, all elements less than 51—leaving

56 65 77 81 82 93 99

Next, the algorithm checks 81—the middle of the remaining elements—to determine whether it matches the search key. The search key (65) is smaller than 81, so 81 is eliminated from consideration along with the elements greater than 81, leaving

56 65 77

After just two tests, the algorithm has narrowed the number of elements to check to three. The algorithm then checks 65, which matches the search key. So the algorithm returns that element's index (9). In this case, the algorithm required just three comparisons to determine whether the array contained the search key. Using a linear search algorithm would have required 10 comparisons.

In this example, we used an array with 15 elements so there would always be an obvious middle array element. With an even number of elements, the middle of the array lies between two elements. We'll implement the algorithm to choose the element with the higher index number.

21.4.1 Implementation

Figure 21.2 implements and demonstrates the binary search algorithm. Throughout the program's execution, we use function template `displayElements` (lines 10–23) to display the portion of the array currently being searched.

```

1 // Fig 21.2: BinarySearch.cpp
2 // Binary search of an array.
3 #include <algorithm>
4 #include <array>
5 #include <format>
6 #include <iostream>
7 #include <random>
8
9 // display array elements from index low through index high
10 template <typename T, size_t size>
11 void displayElements(const std::array<T, size>& items,
12     size_t low, size_t high) {
13
14     for (size_t i{0}; i < items.size() && i < low; ++i) {
15         std::cout << "    "; // display spaces for alignment
16     }
17
18     for (size_t i{low}; i < items.size() && i <= high; ++i) {
19         std::cout << std::format("{} ", items[i]); // display element
20     }
21
22     std::cout << "\n";
23 }
24
25 // perform a binary search on the data
26 template <typename T, size_t size>
27 int binarySearch(const std::array<T, size>& items, const T& key) {
28     int low{0}; // low index of elements to search
29     int high{static_cast<int>(items.size()) - 1}; // high index
30     int middle{ (low + high + 1) / 2}; // middle element
31     int location{-1}; // key's index; -1 if not found
32
33     do { // loop to search for element
34         // display remaining elements of array to be searched
35         displayElements(items, low, high);
36
37         // output spaces for alignment
38         for (int i{0}; i < middle; ++i) {
39             std::cout << "    ";
40         }
41
42         std::cout << " * \n"; // indicate current middle
43
44         // if the element is found at the middle
45         if (key == items[middle]) {
46             location = middle; // location is the current middle
47         }
48         else if (key < items[middle]) { // middle is too high
49             high = middle - 1; // eliminate the higher half
50         }
51         else { // middle element is too low
52             low = middle + 1; // eliminate the lower half
53         }

```

Fig. 21.2 | Binary search of an array. (Part I of 3.)

```
54     middle = (low + high + 1) / 2; // recalculate the middle
55 } while ((low <= high) && (location == -1));
56
57 return location; // return location of key
58 }
59 }
60
61 int main() {
62     // use the default random-number generation engine to produce
63     // uniformly distributed pseudorandom int values from 10 to 99
64     std::random_device rd; // used to seed the default_random_engine
65     std::default_random_engine engine{rd()}; // rd() produces a seed
66     std::uniform_int_distribution<int> randomInt{10, 99};
67
68     std::array<int, 15> arrayToSearch; // create 15-element array
69
70     // fill arrayToSearch with random values
71     for (int &item : arrayToSearch) {
72         item = randomInt(engine);
73     }
74
75     std::sort(arrayToSearch.begin(), arrayToSearch.end());
76
77     // display arrayToSearch's values
78     displayElements(arrayToSearch, 0, arrayToSearch.size() - 1);
79
80     // get input from user
81     std::cout << "\nPlease enter an integer value (-1 to quit): ";
82     int searchKey; // value to locate
83     std::cin >> searchKey; // read an int from user
84     std::cout << "\n";
85
86     // repeatedly input an integer; -1 terminates the program
87     while (searchKey != -1) {
88         // use binary search to try to find integer
89         int position{binarySearch(arrayToSearch, searchKey)};
90
91         // return value of -1 indicates integer was not found
92         if (position == -1) {
93             std::cout << std::format("{} was not found.\n", searchKey);
94         }
95         else {
96             std::cout << std::format("{} was found in position {}.\n",
97                                     searchKey, position);
98         }
99
100        // get input from user
101        std::cout << "\n\nPlease enter an integer value (-1 to quit): ";
102        std::cin >> searchKey; // read an int from user
103        std::cout << "\n";
104    }
105 }
```

Fig. 21.2 | Binary search of an array. (Part 2 of 3.)

```

10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
Please enter an integer value (-1 to quit): 48
10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
*
10 23 27 48 52 55 58
*
48 was found in position 3.

Please enter an integer value (-1 to quit): 92
10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
*
62 63 68 72 75 92 97
*
75 92 97
*
92 was found in position 13.

Please enter an integer value (-1 to quit): 22
10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
*
10 23 27 48 52 55 58
*
10 23 27
*
10
*
22 was not found.

Please enter an integer value (-1 to quit): -1

```

Fig. 21.2 | Binary search of an array. (Part 3 of 3.)

Function Template `binarySearch`

Lines 26–59 define the `binarySearch` function template. Its function parameters are:

- a reference to the array to search, and
- a reference to the search key

Lines 28–30 calculate the low-end index, high-end index and middle index of the portion of the array that the algorithm is currently searching. When `binarySearch` is first called, `low` is 0, `high` is the array's size minus 1, and `middle` is the average of these two values. Line 31 initializes `location` to -1—the value `binarySearch` returns if the search key is not found. Lines 33–56 loop until

- `low` is greater than `high`, indicating that the element was not found, or
- `location` does not equal -1, indicating that the search key was found.

Line 45 tests whether the value in the `middle` element is equal to `key`. If so, line 46 assigns the `middle` index to `location`, the loop terminates, and the function returns `location` to

the caller. Each loop iteration that does not find the search key tests a single value (line 48) and eliminates half of the remaining values in the array (line 49 or 51).

Function main

Lines 64–66 set up a random-number generator for `int` values from 10–99. Lines 68–73 create an array and fill it with random `ints`. Recall that the binary search algorithm requires a sorted array, so line 75 uses `std::sort` to sort `arrayToSearch`'s elements into ascending order. Then, line 78 displays `arrayToSearch`'s sorted contents.

Lines 87–104 loop until the user enters the value -1. For each search key the user enters, the program performs a binary search of `arrayToSearch` to determine whether it contains that search key. First, we show `arrayToSearch`'s contents in ascending order. When the user instructs the program to search for 48, `binarySearch` tests the middle element—60, as indicated by *. The search key is less than 60, so the program eliminates the array's second half and tests the middle element from the array's first half. The search key equals 48, so the program returns the index 3 after performing just two comparisons. The output also shows the results of searching for the values 92 and 22.

21.4.2 Efficiency of Binary Search



In the worst-case scenario, searching a sorted array of 1,023 elements ($2^{10} - 1$) takes **only 10 comparisons** with binary search. The algorithm compares the search key to the middle element of the sorted array. If it's a match, the search is over. Otherwise, if the search key is larger than the middle element, we can eliminate the array's first half from further consideration, and if it's smaller, we can eliminate the array's second half from further consideration. This creates a halving effect—subsequent searches have to search only 511, 255, 127, 63, 31, 15, 7, 3 and 1 elements. The number 1,023 ($2^{10} - 1$) 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—tremendous performance improvements over the linear search. For a one-billion-element array, this is the 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 the base can be omitted for Big O purposes. This results in a Big O of **$O(\log n)$** for a binary search, known as **logarithmic running time** and pronounced as “order $\log n$.”



Checkpoint

- 1 **(Fill-in)** The linear search and the binary search are dramatically different search algorithms. Yet, for each, the worst-case runtime occurs when _____.

Answer: the search key is not found.

- 2 **(Discussion)** It's expensive to sort an array to enable it to be searched efficiently with the binary search algorithm. Mention two circumstances making it worthwhile to do this.

Answer: 1. When the array will be searched a great many times. 2. When the application has strict performance requirements.

3 (Discussion) Compare the worst-case performance of a linear search on an unsorted trillion-element array with the worst-case performance of a binary search on a trillion element sorted array.

Answer: The linear search would have to perform about a trillion comparisons to determine that the search key doesn't match an item in the unsorted array or matches an element close to the array's end. The binary search would have to perform about 40 comparisons to determine that the search key either doesn't match an item in the sorted array or matches an item on the last few comparisons.

21.5 Insertion Sort

Next, let's implement a sorting algorithm. The insertion sort algorithm's first iteration takes the array's second element and, if it's less than the first element, swaps it with the first element—that is, the algorithm **inserts** the second element in front of the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements so all three elements are in order.

First Iteration

Line 27 of Fig. 21.3 declares and initializes the array named `data` with the following values:

```
34  56  4   10   77   51   93   30   5   52
```

Line 36 passes the array to the `insertionSort` function, which receives the array in parameter `items`. The function first looks at `items[0]` and `items[1]`, whose values are 34 and 56, respectively. These two elements are already in order, so the algorithm continues. If they were out of order, the algorithm would swap them.

Second Iteration

In the second iteration, the algorithm looks at the value of `items[2]` (that is, 4). This value is less than 56, so the algorithm stores 4 in a temporary variable and moves 56 one element to the right. The algorithm then determines that 4 is less than 34, so it moves 34 one element to the right. At this point, the algorithm has reached the beginning of the array, so it places 4 in `items[0]`. The array is now:

```
4   34  56  10   77   51   93   30   5   52
```

Third Iteration and Beyond

In the third iteration, the algorithm places the value of `items[3]` (that is, 10) in the correct location with respect to the first four array elements. The algorithm compares 10 to 56 and moves 56 one element to the right because it's larger than 10. Next, the algorithm compares 10 to 34, moving 34 one element to the right. When the algorithm compares 10 to 4, it observes that 10 is greater than 4 and places 10 in `items[1]`. The array is now:

```
4   10  34  56  77   51   93   30   5   52
```

With this algorithm, after the i th iteration, the first $i + 1$ array elements are sorted. However, they may not be in their final locations because the algorithm might encounter smaller values later in the array.

21.5.1 Implementation

Figure 21.3 uses the simple but inefficient **insertion sort** algorithm to sort a 10-element array's values into ascending order. Function template `insertionSort` (lines 8–24) implements the algorithm.

```

1 // Fig. 20.3: InsertionSort.cpp
2 // Sorting an array into ascending order with insertion sort.
3 #include <array>
4 #include <format>
5 #include <iostream>
6
7 // sort an array into ascending order
8 template <typename T, size_t size>
9 void insertionSort(std::array<T, size>& items) {
10     // loop over the elements of the array
11     for (size_t next{1}; next < items.size(); ++next) {
12         T insert{items[next]}; // save value of next item to insert
13         size_t moveIndex{next}; // initialize location to place element
14
15         // search for the location in which to put the current element
16         while ((moveIndex > 0) && (items[moveIndex - 1] > insert)) {
17             // shift element one slot to the right
18             items[moveIndex] = items[moveIndex - 1];
19             --moveIndex;
20         }
21
22         items[moveIndex] = insert; // place insert item back into array
23     }
24 }
25
26 int main() {
27     std::array<int, 10> data{34, 56, 4, 10, 77, 51, 93, 30, 5, 52};
28
29     std::cout << "Unsorted array:\n";
30
31     // output original array
32     for (size_t i{0}; i < data.size(); ++i) {
33         std::cout << std::format("{:>4}", data[i]);
34     }
35
36     insertionSort(data); // sort the array
37
38     std::cout << "\nSorted array:\n";
39
40     // output sorted array
41     for (size_t i{0}; i < data.size(); ++i) {
42         std::cout << std::format("{:>4}", data[i]);
43     }
44
45     std::cout << "\n";
46 }
```

Fig. 21.3 | Sorting an array into ascending order with insertion sort. (Part 1 of 2.)

```

Unsorted array:
 34 56 4 10 77 51 93 30 5 52
Sorted array:
 4 5 10 30 34 51 52 56 77 93

```

Fig. 21.3 | Sorting an array into ascending order with insertion sort. (Part 2 of 2.)

Lines 11–23 in `insertionSort` perform the sorting. In each iteration, line 12 temporarily stores in variable `insert` the value of the element that will be inserted into the array's sorted portion. Line 13 declares and initializes the variable `moveIndex`, which keeps track of where to insert the element. Lines 16–20 loop to locate the correct position where the element should be inserted. The loop terminates when it reaches the array's first element or an element less than the value to insert. Line 18 moves an element to the right, and line 19 decrements the position at which to insert the next element. After the `while` loop ends, line 22 inserts the element into place. When the `for` statement in lines 11–23 terminates, the array's elements are sorted.

Perf 21.5.2 Efficiency of Insertion Sort

Insertion sort's inefficiency becomes apparent when sorting large arrays. The algorithm iterates $n - 1$ times, inserting an element into the appropriate position in the elements sorted so far. For each iteration, determining where to insert the element can require comparing the element to each preceding element—a worst case of $n - 1$ comparisons. Each individual iteration statement runs in $O(n)$ time, so you determine Big O notation for nested statements by multiplying the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each $O(n)$ iteration of the outer loop, there will be $O(n)$ iterations of the inner loop, resulting in a Big O of $O(n * n)$ or $O(n^2)$.



Checkpoint

- I** (Fill-In) The insertion sort algorithm's first iteration takes the array's second element and, if it's less than the first, swaps it with the first. The second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order. After the algorithm's i th iteration, the array's _____ elements will be sorted.

Answer: first $i + 1$.

21.6 Selection Sort

The selection sort algorithm's first iteration selects the smallest element value and swaps it with the first element's value. The second iteration selects the second-smallest element value and swaps it with the second element's value. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element's value, leaving the largest value in the last element. After the i th iteration, the smallest i values will be sorted into increasing order in the first i array elements.

First Iteration

Line 29 in Fig. 21.4 declares and initializes the array named `data` with the following values:

```
34 56 4 10 77 51 93 30 5 52
```

The selection sort first determines the smallest value (4) in the array, located at index 2. The algorithm swaps 4 with the value at index 0 (34), resulting in

```
4 56 34 10 77 51 93 30 5 52
```

Second Iteration

The algorithm then determines the smallest value of the remaining elements (all elements except 4), which is 5, located at index 8. The program swaps the 5 with the 56 at index 1, resulting in

```
4 5 34 10 77 51 93 30 56 52
```

Third Iteration

On the third iteration, the program determines the next smallest value, 10, and swaps it with the value at index 2 (34).

```
4 5 10 34 77 51 93 30 56 52
```

The process continues until the array is fully sorted.

```
4 5 10 30 34 51 52 56 77 93
```

After the first iteration, the smallest element is in the first position; after the second iteration, the two smallest elements are in order in the first two positions and so on.

21.6.1 Implementation

Figure 21.4 uses the simple but inefficient `selection sort` algorithm to sort a 10-element array's values into ascending order. Function template `selectionSort` (lines 8–26) implements the algorithm.

```

1 // Fig. 21.4: SelectionSort.cpp
2 // Sorting an array into ascending order with selection sort.
3 #include <array>
4 #include <format>
5 #include <iostream>
6
```

Fig. 21.4 | Sorting an array into ascending order with selection sort. (Part I of 2.)

```

7 // sort an array into ascending order
8 template <typename T, size_t size>
9 void selectionSort(std::array<T, size>& items) {
10    // loop over size - 1 elements
11    for (size_t i{0}; i < items.size() - 1; ++i) {
12        size_t indexOfSmallest{i}; // will hold index of smallest element
13
14        // loop to find index of smallest element
15        for (size_t index{i + 1}; index < items.size(); ++index) {
16            if (items[index] < items[indexOfSmallest]) {
17                indexOfSmallest = index;
18            }
19        }
20
21        // swap the elements at positions i and indexOfSmallest
22        T hold{items[i]};
23        items[i] = items[indexOfSmallest];
24        items[indexOfSmallest] = hold;
25    }
26 }
27
28 int main() {
29     std::array<int, 10> data{34, 56, 4, 10, 77, 51, 93, 30, 5, 52};
30
31     std::cout << "Unsorted array:\n";
32
33     // output original array
34     for (size_t i{0}; i < data.size(); ++i) {
35         std::cout << std::format("{:>4}", data[i]);
36     }
37
38     selectionSort(data); // sort the array
39
40     std::cout << "\nSorted array:\n";
41
42     // output sorted array
43     for (size_t i{0}; i < data.size(); ++i) {
44         std::cout << std::format("{:>4}", data[i]);
45     }
46
47     std::cout << "\n";
48 }
```

```

Unsorted array:
 34  56   4   10   77   51   93   30   5   52
Sorted array:
   4   5   10   30   34   51   52   56   77   93

```

Fig. 21.4 | Sorting an array into ascending order with selection sort. (Part 2 of 2.)

Function template `selectionSort` performs the sorting in lines 8–26. The loop iterates `size - 1` times. Line 12 declares and initializes the variable `indexOfSmallest`, which stores the index of the smallest element in the unsorted portion of the array. Lines 15–19

iterate over the remaining array elements. For each element, line 16 compares the current element's value to the value at `indexOfSmallest`. If the current element is smaller, line 17 assigns the current element's index to `indexOfSmallest`. When this loop finishes, `indexOfSmallest` contains the index of the array's smallest remaining element. Lines 22–24 then swap the elements at positions `i` and `indexOfSmallest`, using the temporary variable `hold` to store `items[i]`'s value while that element is assigned `items[indexOfSmallest]`.

21.6.2 Efficiency of Selection Sort



The selection sort algorithm iterates $n - 1$ times, each time swapping the smallest remaining element into its sorted position. Locating the smallest remaining element requires $n - 1$ comparisons during the first iteration, $n - 2$ during the second iteration, then $n - 3, \dots, 3, 2, 1$. This results in a total of $n(n - 1)/2$ or $(n^2 - n)/2$ comparisons. In Big O notation, smaller terms drop out, and constants are ignored, leaving a Big O of $O(n^2)$. Can we develop sorting algorithms that perform better than $O(n^2)$?



Checkpoint

- 1 *(Discussion)* Consider the following array, which reflects the result of a selection sort's first pass:

4 56 34 10 77 51 93 30 5 52

What does the second pass do? Show the resulting array.

Answer: The second pass swaps 56 with 5 (the second smallest element), resulting in:

4 5 34 10 77 51 93 30 56 52

- 2 *(Discussion)* The insertion sort and selection sort algorithms both run in $O(n^2)$ time. What program structure do they each have that causes the $O(n^2)$ run time?

Answer: They each have a nested for-loop.

21.7 Merge Sort (A Recursive Implementation)

Merge sort is an efficient sorting algorithm but is conceptually more complex than insertion sort and selection sort. The merge sort algorithm sorts an array by splitting it into two equal-sized sub-arrays, sorting each sub-array then merging them into one larger array. With an odd number of elements, the algorithm creates the two sub-arrays such that one has one more element than the other.

Merge sort performs the merge by looking at each sub-array's first element—the smallest one in each sub-array. Merge sort takes the smallest of these and places it in the first element of the merged sorted array. If there are still elements in the sub-array, merge sort looks at the second element (now the smallest remaining one) and compares it to the first element in the other sub-array. Merge sort continues this process until the merged array is filled. Once a sub-array has no more elements, the merge copies the other array's remaining elements into the merged array.

Sample Merge

Suppose the algorithm has already merged smaller arrays to create sorted arrays A:

4 10 34 56 77

and B:

5 30 51 52 93

Merge sort merges these arrays into a sorted array. The smallest value in A is 4 (located in the zeroth element of A). The smallest value in B is 5 (located in the zeroth element of B). To determine the smallest element in the larger array, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the value of the first element in the merged array. The algorithm continues by comparing 10 (the value of the second element in A) to 5 (the value of the first element in B). The value from B is smaller, so 5 becomes the value of the second element in the larger array. The algorithm continues by comparing 10 to 30, with 10 becoming the value of the third element in the array, and so on.

21.7.1 Implementation

Our merge sort implementation is recursive. The **base case** is a one-element array. Of course, such an array is sorted, so merge sort immediately returns when called with a one-element array. The **recursion step** splits an array of two or more elements into two equal-sized sub-arrays, recursively sorts each sub-array, then merges them into one larger, sorted array. If there is an odd number of elements, one sub-array will be one element larger than the other.

Figure 21.5 implements and demonstrates the merge sort algorithm. Throughout the program's execution, we use function template `displayElements` (lines 9–21) to display the portions of the array currently being split and merged. Function templates `merge` (lines 24–71) and `mergeSort` (lines 74–97) implement the merge sort algorithm. Function `main` (lines 99–123) creates an array, populates it with random integers, executes the algorithm (line 118) and displays the sorted array. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

```

1 // Fig 21.5: MergeSort.cpp
2 // Sorting an array into ascending order with merge sort.
3 #include <array>
4 #include <format>
5 #include <iostream>
6 #include <random>
7
8 // display array elements from index low through index high
9 template <typename T, size_t size>
10 void displayElements(const std::array<T, size>& items,
11     size_t low, size_t high) {
12     for (size_t i{0}; i < items.size() && i < low; ++i) {
13         std::cout << "   "; // display spaces for alignment
14     }
15 }
```

Fig. 21.5 | Sorting an array into ascending order with merge sort. (Part 1 of 5.)

```
16     for (size_t i{low}; i < items.size() && i <= high; ++i) {
17         std::cout << std::format("{} ", items[i]); // display element
18     }
19
20     std::cout << "\n";
21 }
22
23 // merge two sorted subarrays into one sorted subarray
24 template <typename T, size_t size>
25 void merge(std::array<T, size>& items,
26             size_t left, size_t middle1, size_t middle2, size_t right) {
27     size_t leftIndex{left}; // index into left subarray
28     size_t rightIndex{middle2}; // index into right subarray
29     size_t combinedIndex{left}; // index into temporary working array
30     std::array<T, size> combined; // working array
31
32     // output two subarrays before merging
33     std::cout << "merge: ";
34     displayElements(items, left, middle1);
35     std::cout << " ";
36     displayElements(items, middle2, right);
37     std::cout << "\n";
38
39     // merge arrays until reaching end of either
40     while (leftIndex <= middle1 && rightIndex <= right) {
41         // place smaller of two current elements into result
42         // and move to next space in array
43         if (items[leftIndex] <= items[rightIndex]) {
44             combined[combinedIndex++] = items[leftIndex++];
45         }
46         else {
47             combined[combinedIndex++] = items[rightIndex++];
48         }
49     }
50
51     if (leftIndex == middle2) { // if at end of left array
52         while (rightIndex <= right) { // copy in rest of right array
53             combined[combinedIndex++] = items[rightIndex++];
54         }
55     }
56     else { // at end of right array
57         while (leftIndex <= middle1) { // copy in rest of left array
58             combined[combinedIndex++] = items[leftIndex++];
59         }
60     }
61
62     // copy values back into original array
63     for (size_t i = left; i <= right; ++i) {
64         items[i] = combined[i];
65     }
66 }
```

Fig. 21.5 | Sorting an array into ascending order with merge sort. (Part 2 of 5.)

```
67 // output merged array
68 std::cout << "      ";
69 displayElements(items, left, right);
70 std::cout << "\n";
71 }
72
73 // split array, sort subarrays and merge subarrays into sorted array
74 template <typename T, size_t size>
75 void mergeSort(std::array<T, size>& items, size_t low, size_t high) {
76     // test base case; size of array equals 1
77     if ((high - low) >= 1) { // if not base case
78         size_t middle1{(low + high) / 2}; // calculate middle of array
79         size_t middle2{middle1 + 1}; // calculate next element over
80
81         // output split step
82         std::cout << "split:   ";
83         displayElements(items, low, high);
84         std::cout << "      ";
85         displayElements(items, low, middle1);
86         std::cout << "      ";
87         displayElements(items, middle2, high);
88         std::cout << "\n";
89
90         // split array in half; sort each half (recursive calls)
91         mergeSort(items, low, middle1); // first half of array
92         mergeSort(items, middle2, high); // second half of array
93
94         // merge two sorted arrays after split calls return
95         merge(items, low, middle1, middle2, high);
96     }
97 }
98
99 int main() {
100     // use the default random-number generation engine to produce
101     // uniformly distributed pseudorandom int values from 10 to 99
102     std::random_device rd; // used to seed the default_random_engine
103     std::default_random_engine engine{rd()}; // rd() produces a seed
104     std::uniform_int_distribution<int> randomInt{10, 99};
105
106     std::array<int, 10> data; // create array
107
108     // fill data with random values
109     for (int &item : data) {
110         item = randomInt(engine);
111     }
112
113     // display data's values before mergeSort
114     std::cout << "Unsorted array:\n";
115     displayElements(data, 0, data.size() - 1);
116     std::cout << "\n";
117
118     mergeSort(data, 0, data.size() - 1); // sort the array data
119 }
```

Fig. 21.5 | Sorting an array into ascending order with merge sort. (Part 3 of 5.)

```
120 // display data's values after mergeSort
121 std::cout << "Sorted array:\n";
122 displayElements(data, 0, data.size() - 1);
123 }
```

```
Unsorted array:
30 47 22 67 79 18 60 78 26 54

split:   30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                      18 60 78 26 54

split:   30 47 22 67 79
          30 47 22
                      67 79

split:   30 47 22
          30 47
                      22

split:   30 47
          30
                      47

merge:   30
          47
          30 47

merge:   30 47
          22
          22 30 47

split:           67 79
          67
          79

merge:           67
          79
          67 79

merge:   22 30 47
          67 79
          22 30 47 67 79

split:           18 60 78 26 54
          18 60 78
                      26 54

split:           18 60 78
          18 60
                      78

split:           18 60
          18
          60
```

Fig. 21.5 | Sorting an array into ascending order with merge sort. (Part 4 of 5.)

```

merge:          18
                60
                18 60

merge:          18 60
                78
                18 60 78

split:          26 54
                26
                54

merge:          26
                54
                26 54

merge:          18 60 78
                26 54
                18 26 54 60 78

merge:          22 30 47 67 79
                18 26 54 60 78
                18 22 26 30 47 54 60 67 78 79

Sorted array:
18 22 26 30 47 54 60 67 78 79

```

Fig. 21.5 | Sorting an array into ascending order with merge sort. (Part 5 of 5.)

Function mergeSort

The `merge` function is defined before `mergeSort`, so `mergeSort` can call it, but let's discuss `mergeSort` first. The recursive `mergeSort` function (lines 74–97) receives as parameters the array to sort and the `low` and `high` indices of the range of elements to sort. Line 77 tests the base case. If the `high` index minus the `low` index is 0, the sub-array has one element, so the function simply returns. If the difference between the indices is greater than or equal to 1, the function splits the array in half—lines 78–79 determine the split point. Next, line 91 recursively calls `mergeSort` on the array's first half, and line 92 recursively calls `mergeSort` on the array's second half. When these two calls return, each half is sorted. Line 95 calls `merge` (lines 24–71) on the two halves to combine the sorted sub-arrays into one larger sorted array.

Function merge

Lines 40–49 in `merge` loop until the program reaches the end of either sub-array. Line 40 tests which element at the beginning of the two sub-arrays is smaller. If the element in the left sub-array is smaller or both are equal, line 44 places it in position in the combined array. If the element in the right sub-array is smaller, line 47 places it in position in the combined array. When the `while` loop completes, one entire sub-array is in the combined array, but the other sub-array still contains data. Line 51 tests whether the left sub-array has reached the end. If so, lines 52–54 fill the combined array with the elements of the right sub-array. If the left sub-array has not reached the end, then the right sub-array must have reached the end, and lines 57–59 fill the combined array with the elements of the left sub-array. Finally, lines 63–65 copy the combined array into the original array.



21.7.2 Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort—although that may be difficult to believe when looking at Fig. 21.5’s busy output. Consider the non-recursive first `mergeSort` call (line 118). This results in

- two recursive `mergeSort` calls with sub-arrays that are each approximately half the original array’s size, and
- a single call to `merge`.

The `merge` call requires, at worst, $n - 1$ comparisons to fill the original array, which is $O(n)$. (Recall that each array element is chosen by comparing one element from each sub-array.) The two `mergeSort` calls result in

- four more recursive `mergeSort` calls, each with a sub-array approximately one-quarter the size of the original array, and
- two calls to `merge`.

These two calls to `merge` each require, at worst, $n/2 - 1$ comparisons for a total of $O(n)$ comparisons. This process continues with each `mergeSort` call generating two additional `mergeSort` calls and a `merge` call until the algorithm has split the array into one-element sub-arrays. At each level, $O(n)$ comparisons are required to merge the sub-arrays. Each level splits the arrays in half, so doubling the array size requires one more level and quadrupling the array size requires two more levels. This pattern is logarithmic, resulting in $\log_2 n$ levels for a total efficiency of $O(n \log n)$.

21.7.3 Summarizing Various Algorithms’ Big O Notations

The following diagram summarizes the searching and sorting algorithms we cover in this chapter and lists the Big O for each.

Algorithm	Location	Big O
<i>Searching Algorithms</i>		
Linear search	Section 21.3	$O(n)$
Binary search	Section 21.4	$O(\log n)$
Recursive linear search	Exercise 21.8	$O(n)$
Recursive binary search	Exercise 21.9	$O(\log n)$
<i>Sorting Algorithms</i>		
Insertion sort	Section 21.5	$O(n^2)$
Selection sort	Section 21.6	$O(n^2)$
Merge sort	Section 21.7	$O(n \log n)$
Bubble sort	Exercises 21.5–21.6	$O(n^2)$
Quicksort	Exercise 21.10	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Common Big O Notations

The following table lists various common Big O categories and several 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 should 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
1,000	1	3	1,000	3,000	10^6
1,000,000	1	6	1,000,000	6,000,000	10^{12}



Checkpoint

1 *(Discussion)* The recursive merge sort algorithm sorts an array by splitting it into two equal-size sub-arrays, sorting each sub-arrays, then merging them into one larger array. The sub-arrays in merge sort are not sorted with sorts we've covered, such as the bubble sort, selection sort or insertion sort. Explain how the sub-arrays are sorted in merge sort.
Answer: If the array has an even number of elements, the merge sort splits the array into two equal-size halves. If the array has an odd number of elements, one “half” has one more element than the other “half.”

Each half is then recursively split into two smaller halves. This halving process continues until each half contains only one element. This is the base case of the recursion because a one-element array is sorted.

Next, those individual elements are merged into two-element sorted sub-arrays based on their values. So, in answer to the question, the **sorting is actually trivial**. As the algorithm’s recursion unwinds, merge sort keeps merging smaller sorted sub-arrays to form larger sorted sub-arrays. Every merge of two sorted sub-arrays results in a larger sorted sub-array that’s about double the size of the ones being merged. The last merge results in a sorted version of the original array.

2 *(Discussion)* Here are the first three lines of our merge sort example’s output:

```
split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                  18 60 78 26 54
```

and here are the last three lines of the output:

```
merge:    22 30 47 67 79
          18 26 54 60 78
          18 22 26 30 47 54 60 67 78 79
```

Compare these parts of the output. How are your observations consistent with how the merge sort works?

Answer: 1. In the **final merge phase**, each sub-array corresponds to one of the unsorted sub-arrays produced by the algorithm's **first split pass**. 2. Each sub-array entering the final merge phase is **sorted**. 3. The 10-element **final merged array** contains the same elements as the original unsorted array that entered the first split pass. 4. The 10-element **final merged array** is, in fact, **sorted**. The recursive merge sort splits the original unsorted array into smaller and smaller pieces until they are down to single-element pieces, which it ultimately merges to form the smallest sorted pieces of the original array. It keeps merging those sorted pieces to form larger and larger sorted pieces until it finally merges the two—now sorted—halves of the original unsorted array to form the final sorted array.

21.8 Wrap-Up

This chapter discussed searching and sorting data. We began by introducing Big O notation to express the efficiency of algorithms by measuring their worst-case run times. Throughout the chapter, you saw that Big O is useful for comparing algorithm performance so you can choose the most efficient one.

Next, we discussed searching. We presented the simple but inefficient linear search algorithm. Then, we presented the binary search algorithm, which is faster but more complex than linear search.

Finally, we discussed sorting data. You learned two simple but inefficient sorting techniques—insertion sort and selection sort. Then, we presented the recursive merge sort algorithm, which is more efficient than either the insertion sort or the selection sort but more complex to program.

Thanks for reading *C++ How to Program: An Objects-Natural Approach, 11/e*. We hope that you enjoyed the book and that you found it entertaining and informative. Most of all we hope you feel empowered to apply the technologies you've learned to the challenges you'll face in your career.

Exercises

21.1 Fill in the blanks in each of the following statements:

- A selection sort application would take approximately _____ times as long to run on a 128-element array as on a 32-element array.
- The efficiency of merge sort is _____.

21.2 What key aspect of both the binary search and the merge sort accounts for the logarithmic portion of their respective Big Os?

21.3 In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

21.4 In the text, we say that after the merge sort splits the array into two sub-arrays, it then sorts these two sub-arrays and merges them. Why might someone be puzzled by our statement that “it then sorts these two sub-arrays”?

21.5 (*Bubble Sort*) In this exercise, you'll implement the **bubble sort algorithm**, another simple but inefficient sorting algorithm. It's called bubble sort because smaller values

gradually “bubble” their way to the top of the array (that is, toward the first element) like air bubbles rising in water. It’s also called a sinking sort, as the larger values sink to the bottom (end) of the array. The technique uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the array.

The first pass compares the first two element values and swaps them if necessary. It then compares the second and third element values in the array and swaps them if necessary. This continues until the end of this pass compares the last two element values in the array and swaps them if necessary. After one pass, the largest value will be in the last element. After two passes, the largest two values will be in the last two elements and so on. Implement and test the bubble sort algorithm.

21.6 (Enhanced Bubble Sort) Make the following modifications to improve the performance of the bubble sort you developed in Exercise 21.5:

- After the first pass, the largest value is guaranteed to be located in the highest-numbered array element; after the second pass, the two highest values are “in place,” and so on. Instead of making $n - 1$ comparisons for an n -element array on every pass, modify the bubble sort to make only the $n - 2$ necessary comparisons on the second pass, $n - 3$ on the third pass, and so on.
- The data in the array may already be in the proper order or near-proper order, so why make $n - 1$ passes (of a n -element array) if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If none have been made, the data is sorted, so the program should terminate. If swaps were made, at least one more pass is needed.

21.7 (Bucket Sort) A **bucket sort** begins with a one-dimensional array of positive integers to be sorted and a two-dimensional array of integers with rows indexed from 0 to 9 and columns indexed from 0 to $n - 1$, where n is the number of values to be sorted. Each row of the two-dimensional array is referred to as a **bucket**. Write a class named `BucketSort` containing a function called `sort` that operates as follows:

- Place each value of the one-dimensional array into a row of the bucket array based on the value’s “ones” (rightmost) digit. For example, 97 is placed in row 7, 3 is placed in row 3, and 100 is placed in row 0. This procedure is called a *distribution pass*.
- Loop through the bucket array row by row, and copy the values back to the original array. This procedure is called a *gathering pass*. The new order of the preceding values in the one-dimensional array is 100, 3 and 97.
- Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second (tens digit) pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit), and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional array is 100, 3 and 97. On the third (hundreds digit) pass, 100 is placed in row 1, 3 is placed in row 0, and 97 is placed in row 0 (after the 3). After this last gathering pass, the original array is in sorted order.

Note that the two-dimensional array of buckets is 10 times the length of the integer array being sorted. This sorting technique performs better than a bubble sort but requires much more memory—the bubble sort requires space for only one additional ele-

ment of data. This comparison is an example of the space–time trade-off: The bucket sort uses more memory than the bubble sort but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly swap the data between the two bucket arrays.

21.8 (Recursive Linear Search) Modify Fig. 21.1 to define a linear search function named `recursiveLinearSearch` that performs a recursive linear search of an array. The function should receive the array, the search key and starting index as arguments. If the search key is found, return its index in the array; otherwise, return `-1`. Each call to the recursive function should check one element value in the array.

21.9 (Recursive Binary Search) Modify Fig. 21.2 to define a binary search function named `recursiveBinarySearch` that performs a recursive binary search of an array. The function should receive the array, the search key, starting index and ending index as arguments. If the search key is found, return its index in the array. If the search key is not found, return `-1`.

21.10 (Quicksort) The recursive sorting technique called quicksort uses the following basic algorithm for a one-dimensional array of values:

- Partitioning Step:* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element’s value, and all values to the right of the element in the array are greater than the element’s value—we show how to do this below). We now have one value in its proper location and two unsorted sub-arrays.
- Recursion Step:* Perform the *Partitioning Step* on each unsorted sub-array.

Each time *Step 1* is performed on a sub-array, another element is placed in its final location of the sorted array, and two unsorted sub-arrays are created. When a sub-array consists of one element, it must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each sub-array? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

Starting from the rightmost element of the array, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is **12**, so **37** and **12** are swapped. The values now reside in the array as follows:

12 2 6 4 89 8 10 **37** 68 45

Element **12** is in italics to indicate that it was just swapped with **37**.

Starting from the left of the array but beginning with the element after **12**, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is **89**, so **37** and **89** are swapped. The values now reside in the array as follows:

12 2 6 4 **37** 8 10 89 68 45

Starting from the right but beginning with the element before 89, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The values now reside in the array as follows:

12 2 6 4 10 8 37 89 68 45

Starting from the left but beginning with the element after 10, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted sub-arrays. The sub-array with values less than 37 contains 12, 2, 6, 4, 10 and 8. The sub-array with values greater than 37 contains 89, 68 and 45. The sort continues with both sub-arrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write a recursive `quickSort` function to sort an integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function `partition` should be called by `quickSort` to perform the partitioning step.

Glossary

!

! logical negation operator

“Reverses” a condition’s meaning.

/

// single-line comment

This notation indicates that the remaining text on a line of code is a comment.

[

[[fallthrough]]

Tells the compiler that “falling through” to the next case in a switch statement is the correct behavior

*

*** operator**

The unary version of this is known as the indirection operator or dereferencing operator.

&

&& logical AND operator

Enables a program to ensure that both of two conditions are true before choosing a certain path of execution.

#

#include preprocessing directive

A message to the C++ preprocessor, which the compiler invokes before compiling the program. Notifies the preprocessor to include the contents of a header file in the program.

<

<< stream insertion operator

Use this with `std::cout` to insert a value in the standard output stream.

<iostream> input/output stream header

This file contains 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.

<numbers>

This new C++20 header standardizes some mathematical constants commonly used in many scientific and engineering applications.

<random>

The element of chance can be introduced into your applications with features from this header.

<string> header

Include this in programs that use the C++ standard library’s `string` class.

=

= 0

A pure specifier indicating that a given `virtual` function does not provide an implementation.

= default

Tells the compiler to generate a special member function.

L

|| logical OR operator

Enables a program to ensure that either or both of two conditions are true before choosing a certain path of execution.

A

abstract base class

A class that cannot be instantiated but defines a common public interface for the classes derived from it in a class hierarchy.

access function

Can read or display data, but not modify it.

addition compound assignment operator, +=

Adds its right operand’s value to the left-side variable’s value, then stores the result in the left-side variable.

address operator, &

A unary operator that obtains the memory address of its operand.

aggregate type

A built-in array, an `array` object or an object of a class that does not have user-declared constructors, `private` or `protected` non-static data members, `virtual` functions, or `private`, `protected` or `virtual` base classes.

algorithm

A procedure for solving a problem in terms of the actions to execute and the order in which these actions execute.

Android

The most widely used mobile and smartphone operating system. It is based on the Linux kernel, the Java programming language and now, the open-source Kotlin programming language. It is open-source and free.

append**argument coercion**

Forcing arguments to the appropriate types specified by the parameter declarations.

arithmetic operators

Used to perform calculations, such as addition (+), subtraction (-), multiplication (*), division (/) and remainder (%).

arithmetic overflow

Adding integers could result in a value too large to store in an int variable.

assembly language

Programming in machine language was too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations.

assertion

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.

assignment operator =

Used to store a value in a variable.

asterisk, *, arithmetic operator

The multiplication operator.

asynchronous events

Occur in parallel with and independent of the program's flow of control.

at member function of class**string**

Throws an exception if its argument is an invalid out-of-bounds index for a string. If the index is valid, the function returns the character at the specified location as a modifiable lvalue or a nonmodifiable lvalue.

B**backslash, **

When this escape character is encountered in a string, the next character is combined

bad_alloc exception

Occurs when the new operator is unable to find sufficient space in memory for a dynamically allocated object.

banker's rounding

Fixes the bias in half-up rounding.

base-class initializer

A member initializer that passes arguments to a base-class constructor.

Big O notation

Characterizes how hard an algorithm may have to work to solve a problem.

binary operator

An operator like + that has two operands.

binary search

More efficient than linear search but requires that the array first be sorted.

bit

Digit that can assume one of two values, 0 or 1—and is a computer's smallest data item.

block

Statements in a pair of braces such as a control statement's or function's body form one of these.

boolalpha

Tells an output stream to display condition values as the words false or true.

bool data type

For variables that can hold only the values true or false.

Boost C++ libraries

Serve as a "breeding ground" for new capabilities that might eventually be incorporated into the C++ standard libraries.

Boost Multiprecision library

Includes pre-built classes that can be used to perform precise monetary calculations.

Boost Multiprecision library's**cpp_dec_float_50 class**

Can create objects that represent exact monetary amounts like 123.02 and perform to-the-penny arithmetic without representationlal error.

Boost Multiprecision open source library's cpp_int

A preexisting C++ open-source class for creating and manipulating huge integers.

break statement

Executing this in a while, for, do...while or switch causes immediate exit from that statement—execution continues with the first statement after the control statement.

C

C++

Provides many features that “spruce up” C, but more importantly, it includes capabilities for object-oriented programming inspired by the Simula simulation programming language.

camel case

A naming convention in which identifiers begin with a lowercase letter, and every word in the name after the first begins with a capital letter.

capacity of a string

The number of characters that can be stored in a string before it must allocate more memory to store additional characters.

capture a variable

Enables a lambda expression to specify in its introducer that the compiler should allow a local variable from the enclosing scope to be used in a lambda expression’s body.

capturing a variable

Occurs when a lambda introducer contains a variable from the lambda’s enclosing scope. Enables the variable to be used in the lambda’s body.

caret, ^, in a regular expression

When a custom character class starts with this, the class matches any character that’s not specified.

case-sensitive

C++ treats uppercase and lowercase letters as different.

catch(. . .) handler

Can catch all exception types thrown in a try block. Primarily used to perform recovery that does not depend on the exception type. The caught exception can be rethrown to alert enclosing try statements.

chain of constructor calls

Instantiating a derived-class object begins this to initialize the object.

chain of destructor calls

This is initiated when a derived-class object goes out of scope to clean up the object’s resources.

character class

A regular expression escape sequence that matches one character.

character constant

An integer value represented as a character in single quotes.

ciphertext

Encrypted text.

class hierarchy

Inheritance relationships naturally form one of these. A base class exists in this kind of relationship with its derived classes.

class invariant

Requires a class’s data members to have specific values or ranges of values.

class scope

Qualifying a function definition with a class name and the scope resolution operator (::) indicates that it’s part of this and its name is known to other class members.

client of a class

Any code that calls member functions on an object of a class.

command-line arguments

Often passed to applications as they begin executing to specify configuration options, file names to process and more.

comma operator

This operator guarantees that its operands evaluate left-to-right.

comment

You insert these to document your programs and help other people read and understand them. They do not cause the computer to perform any action when the program is run—they’re ignored by the C++ compiler.

compilers

These convert high-level-language source code into machine language.

composition

A class can have objects of other classes as members (also called a *has-a* relationship).

composition

Also called a *has-a* relationship.

compound statement or block

Multiple statements enclosed in a pair of braces, { }.

concatenating, chaining or cascading stream insertion operations

Enables one statement to display multiple values using cout and several << operators.

concrete class

A class that can be used to instantiate objects.

const

Preceding a variable definition by this keyword is a good practice for any variable that should not change once initialized. This enables the compiler to report errors if you accidentally modify the variable.

constant run time

An algorithm that is independent of the number of array elements is said to have this.

constructor

Specifies how to initialize objects of its class.

constructor injection

A technique in which a constructor receives a pointer or reference to another object and stores it in the object being initialized.

contextual conversion

In contexts that require `bool` values, such as control statement conditions, C++ can invoke an object's `bool` conversion operator implicitly.

continue statement

Executing this in a `while`, `for` or `do...while` skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

control-statement nesting

One control statement appears inside another.

control-statement stacking

Connect the exit point of one control statement to the entry point of the next.

conversion constructor

Can turn objects of other types (including fundamental types) into objects of a particular class.

conversion operator

Converts an object of a class to another type. Must be a non-static member function.

copy-and-swap idiom**copy constructor**

Copies each member of its argument object the corresponding member of the new object.

copy semantics

How to copy an object when it is passed by value to a function, returned by value from a function or assigned to another object.

counter-controlled iteration

Uses a variable to control the number of times a set of statements will execute.

C-strings

Pointer-based string, as inherited from C++'s precursor programming language.

CSV (comma-separated values)

Popular file format datasets used in big data, machine learning and deep learning. Here, we'll demonstrate reading from a CSV file.

D**dangling**

References to undefined variables are of this kind. This is a logic error for which compilers often issue a warning.

dangling pointer

A pointer that points to memory that no longer represents a valid object.

database

A collection of data organized for easy access and manipulation.

data mining

The process of searching through extensive collections of data, often big data, to find insights that can be valuable to individuals and organizations.

data persistence

Files are used for this—permanent data retention.

declaration

Specifies the name and type of a variable.

deep copies

Copies an object and any other objects its members point to or refer to.

default argument

It's common for a program to invoke a function from several places with the same argument value for a particular parameter. In such cases, you can specify that such a parameter has this kind of argument—a value the compiler will insert for you to be passed to that parameter.

default assignment operator

This is generated for a class by the compiler. It copies each data member of the right operand into the same data member in the left operand.

definition

A variable declaration that also reserves memory.

delegating constructor

Calls another constructor in the same class to perform its work.

dependency injection

Constructor injection and property injection are both forms this.

dereferencing a pointer

Applying `*` to a pointer to access the data it points to.

descriptive statistics

Part of getting to know a dataset—including a dataset's count, average, minimum, maximum and median values, among others.

designated initializers

Specifies by name which subset of the data members to initialize in a `struct` object.

destructor

Performs termination housekeeping on an object before the object's memory is reclaimed for later use.

devirtualization

A compile-time optimization enabling the compiler to determine which virtual function to call.

diamond symbol

In a UML activity diagram, this indicates that a decision is to be made.

dot operator, .

You specify an object's name, followed by this operator, a member function name and a set of parentheses to call a member function.

double-precision floating-point number

Most of today's systems store these in eight bytes of memory with approximately 15 significant digits.

do...while iteration statement

Tests the loop-continuation condition after executing the loop's body, so the body always executes at least once.

driver program

A program with a main function that calls objects' member functions, without

DRY

"Don't repeat yourself."

dynamic binding

Also called late binding, this process chooses the appropriate function to call at execution time.

dynamic memory management

Allocating and deallocating memory for objects and for arrays of any built-in or

E**empty member function of class string**

Returns true if a string does not contain any characters—that is, the length of the string is 0.

empty string

A string object that does not contain characters (that is, "").

encapsulate

Classes wrap attributes and member functions into objects created from those classes—an object's attributes and member functions are intimately related.

end-of-file indicator

A system-dependent keystroke combination that indicates there's no more data to input.

ends_with member function of class string

Returns true if a string terminates with a specified substring; otherwise, returns false.

exception

Indicates a problem that occurs during a program's execution.

exception handling

Helps you write robust, fault-tolerant programs.

exception safety guarantees

When you design your code you decide what level of these you'll make from "none" to "strong".

exit function

Used to terminate a program when a fatal unrecoverable error occurs.

exponential complexity

Problems of this nature can humble even the world's most powerful computers.

extensible programming language

C++ is one of these—each class you create becomes a new type you can use to create objects.

extensible systems

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.

F**fail-fast**

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.

fatal logic error

Causes a program to fail and terminate prematurely.

fatal runtime error

Causes a program to terminate immediately without having successfully performed its jobs.

field

A group of characters or bytes that conveys meaning.

file

Some operating systems view this simply as a sequence of bytes—any organization of these bytes, such as into records, is a view created by the application programmer.

file-position pointer

Represents the byte number of the next byte in the file to be read or written.

final state

The solid circle surrounded by a hollow circle at the bottom of a UML activity diagram represents this.

first and last member functions of a span

Return spans representing the specified number of elements at the beginning or end of a span.

floating-point numbers

A number like 118.45 containing a decimal point.

FLOPS

Floating-point operations per second.

for iteration statement

Specifies the counter-controlled-iteration details in a single line of code.

format function

C++20 introduced powerful new string-formatting capabilities via this.

format specifier >7.2f

Says that a floating-point value (f) being formatted should be right-aligned (>) in a field width of 7 character positions with two digits of precision (.2) to the right of the decimal point.

free store

A region of memory assigned to each program for storing dynamically allocated objects.

friend function

Has access to a class's public and non-public members.

front and back member**functions of a span**

Return a span's first and last elements of a span.

function-call operator

The () operator, which is powerful because functions can take an arbitrary number of comma-separated parameters.

function header

The function's first line is also known as this.

function overloading

Functions of the same name can be defined, as long as they have different signatures.

function signature

A function's name and its parameter types together are known as this.

function templates

Overloaded functions typically perform similar operations on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently with these.

fundamental types

Types built into C++, such as int, double, char and long.

G**generic programming**

Programming with templates is also known as this.

get pointer

Indicates the byte number in the file from which the next input should be read.

gigabyte

Approximately one billion bytes.

GitHub

Provides tools for managing open-source software-development projects. It has millions of them underway.

global functions

Functions like main that are not member functions.

global namespace

An identifier declared outside any function or class has this scope.

golden ratio

The ratio of successive Fibonacci numbers converges on a constant value of 1.618, which humans tend to find aesthetically pleasing.

goto statement

Allows you to specify a transfer of control to one of a wide range of destinations in a program.

greedy quantifier

Describes a regular expression quantifier that matches as many characters as possible.

grouping

Also called associativity, this specifies whether C++ applies certain operators left-to-right or right-to-left.

guard condition

Each transition arrow emerging from a UML decision symbol has one of these.

H**halving**

On every pass, if the binary search has not yet found the search key, it does this to eliminate from contention a portion of the remaining elements in the array, so it can find the search key far faster, on average, than the linear search can.

hardware

Computers and related devices are classified as this.

has-a relationship

Represents composition in which an object contains as members one or more objects of other classes.

header with a .h filename**extension**

It's customary to place a reusable class definition in this type of file then include this file wherever you need to use the class.

high-level languages

To speed the programming process, these were developed in which single statements could accomplish substantial tasks. These allow you to write instructions that look almost like everyday English and contain common mathematical notations.

HTTPS (HyperText Transfer Protocol Secure)

Most websites now use this to encrypt and decrypt your Internet interactions.

HTTPS protocol

Used by most websites to encrypt and decrypt your web interactions.

hypot

This two-argument function calculates a right triangle's hypotenuse.

|

identifier

A series of characters consisting of letters, digits and underscores (_) that does not begin with a digit and is not a keyword.

if selection statement

Performs an action (or group of actions) if a condition is true or skips it if the condition is false.

if statement

Allows a program to perform different actions, based on whether a condition is true or false.

if...else double-selection statement

Allows you to specify an action to perform when a condition is true and another action when that condition is false.

if...else selection 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.

implementation inheritance

Primarily used to define closely related classes with many of the same data members and member-function implementations.

in-class initializer

Sets the initial value of a data member where the class declares it.

include guard

Prevents a header's contents from being included in the same source-code file more than once.

indentation

Helps make a program's structure stand out, so the program is easier to read.

indirect base class

A class that's two or more levels up the class hierarchy from its derived classes.

indirection

The process of referencing a value through a pointer.

infinite loop

Not providing in a `while` statement's body an action that eventually causes the condition to become `false` results this logic error.

inheritance

Used to create classes that absorb existing classes' capabilities, then customize or enhance them.

inline function

The compiler can expand these in place of function calls to help reduce function-call overhead.

insertion sort algorithm

An easy-to-implement but inefficient sorting algorithm. The algorithm's first iteration takes the array's second element and, if it's less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements so all three elements are in order. At the i th iteration of this algorithm, the first i elements in the original array will be sorted.

integer

A whole number value, such as 3, -22, 0 or 12345. Variables that store whole numbers can be declared with type `int`.

integer division

Division in which the numerator and the denominator are integers; produces an

integer division

In this kind of calculation, any fractional part of the result is truncated.

integrated development environments—IDEs

Provide tools that support the software development process, including editors for writing and editing programs and debuggers for locating logic errors—errors that cause programs to execute incorrectly.

interface inheritance

Hierarchies designed for this kind of inheritance tend to have concrete definitions of their functionality lower in the hierarchy.

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.

interpreters

Execute high-level language programs directly, avoiding compilation delays, but your code runs slower than compiled programs.

invalid_argument exception

An exception used to notify client code it passed incorrect values a function.

invariant

An invariant is a condition that should always be true in your code—that is, a condition that never changes.

ios::app

File open mode that appends all output to the end of a file without modifying any data already in the file.

ios::ate

File open mode that opens a file for output and moves to the end of the file. Used to append data to a file. Data can be written anywhere in the file.

ios::beg

Seek direction for positioning the file-position pointer relative to the beginning of a stream.

ios::binary

File open mode that opens a file for binary (i.e., non-text) input or output.

ios::cur

Seek direction for positioning the file-position pointer relative to the current position in a stream.

ios::end

Seek direction for positioning the file-position pointer backward relative to the end of a stream.

ios::in

File open mode that opens a file for input.

ios::out

File open mode that opens a file for output.

ios::trunc

File open mode that discards the file's contents. This is the default action for `ios::out`.

IP address

Each Internet-connected device has a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

is-a relationship

Represents an inheritance relationship in which a derived-class object also can be treated as an object of its base-class type.

iteration statements

Also called repetition statements or looping statements.

K**Keywords**

Words reserved by C++ for a specific use.

L**lazy quantifier**

Describes a regular expression quantifier that matches as few characters as possible.

left brace, {

Begins each function's body, which contains the instructions the function performs.

lexicographically

Character comparisons are performed this way, using the integer values of each character. For example, 'A' has the value 65 and 'a' has the value 97, so "Apple" would be considered less than "apple".

linear run time

An $O(n)$ algorithm is said to have this.

linear search

Determines whether an unsorted array contains a specified search key.

linear search algorithm

This runs in $O(n)$ time.

Linux operating system

Operating system that is among the greatest successes of the open-source movement.

local variable

A variable declared in a block – it can be used only from the line of its declaration to the closing right brace of the block.

logical operators

Enable you to combine simple conditions.

logic error

Such as an incorrect calculation, has its effect at execution time.

looping statements

Performs a statement or group of statements repeatedly while a continuation condition remains true.

lvalues

Variable names are these because they can be used on the left side of an assignment.

M**machine language**

Generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time.

main function

This is a part of every C++ program and is where each C++ program begins executing.

make_unique function template

To initialize a `unique_ptr`, call this function template, which allocates dynamic memory with operator `new`, then returns a `unique_ptr` to that memory.

mathematical special functions

C++17 added scores of these to the `<cmath>` header for the engineering and scientific communities.

measures of central tendency

The average and median are examples of these—each is a way to produce a single value, which is, in some sense, typical of the others.

member-initializer list

This is preceded by a colon (:) and placed between a constructor's parameter list and the left brace that begins the constructor's body. It initializes the specified data members of the class.

memory

Also called primary memory or RAM (Random Access Memory).

memory leak

A potential problem with dynamically allocated memory—forgetting to release memory that's no longer needed.

merge sort

Sorts an array by splitting it into two equal-sized subarrays, sorting each subarray then combining them into one larger sorted array.

Moore's law

The name for the fact that, for decades, computer processing power has approximately doubled inexpensively every couple of years.

move semantics

Eliminate costly, unnecessary copies of objects that are about to be destroyed.

multi-core processors

These implement multiple processors on a single integrated circuit chip. Such processors can perform many operations simultaneously.

multiline comment

A comment enclosed in the delimiters /* and */. Such comments can span several lines in a program.

multiple inheritance

A derived class that inherits from two or more base classes uses this kind of inheritance.

multithreaded applications

Taking full advantage of multi-core architecture requires writing these.

N**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.

narrow_cast

If you must perform an explicit narrowing conversion, the C++ Core Guidelines recommend using this type of cast from the Guidelines Support Library (GSL).

narrowing conversions

For fundamental-type variables, braced initializers prevent these operations that could result in data loss.

nested or embedded parentheses

Expressions in these evaluate first; they are said to be at the "highest level of precedence."

newline

Represented by the escape sequence \n.

noexcept

Indicates that a function does not throw any exceptions and does not call any functions that throw exceptions.

nondeterministic random numbers

Random-numbers that cannot be predicted.

nonfatal logic error

Allows a program to continue executing but causes it to produce incorrect results.

null character, '\0'

Marks where a C-string terminates in memory.

nullptr

A pointer with this value "points to nothing."

O**O(log n)**

The Big O notation for logarithmic runtime.

objects natural

Educational approach that enables you to conveniently create and program powerful

off-by-one error

If you intend to loop 10 times but incorrectly specify a loop-continuation condition or an initial loop counter value, the loop might iterate only nine times causing this common logic error.

OpenAI

The creators of ChatGPT and Dall-E 2, which have each generated enormous interest in artificial intelligence.

operating system

Software that makes using computers more convenient for users, software developers and system administrators. Provides services that allow applications to execute safely, efficiently and concurrently with one another.

operator function

Once you define one of these for a given operator and your custom class, that operator has meaning appropriate for objects of your class.

operator overloading

“Teaches” C++ how to use existing operators with objects of new class types.

operator overloading

Enables C++’s existing operators to work with custom class objects.

override a base class virtual function

When a derived class redefines how a base-class function works to make it specific to derived-class objects.

override keyword

Tells the compiler to check whether the base class has a `virtual` member function with the same signature.

P**parameterized stream manipulator**

A stream manipulator that requires an argument to perform its task.

parameter list

Typically, a function definition’s first line contains its return type, function name and parentheses containing this, which specifies any additional information the function needs to perform its task.

pass-by-reference

When a variable is passed this way, the caller gives the called function the ability to access that variable in the caller directly and to modify the variable.

persistent

This describes information on secondary storage devices that is preserved even when the computer’s power is turned off.

pointer arithmetic

Adding an integer to or subtracting an integer from a pointer to move to a different location in a contiguous range of elements, such as an array.

postcondition

Something that is true after a function successfully returns. Used to describe guarantees

precondition

Something that must be true when a function is invoked. Used to describe constraints on function parameters and any other expectations the function has just before it begins executing. If these are not met, then the function’s behavior is undefined.

predicate function

Tests whether a condition is `true` or `false` and returns the result.

private

An access specifier indicating that a data member or member function is not available outside the class.

programs

These guide the computer through ordered actions specified by people called computer programmers.

prompt

Message that directs the user to take a specific action.

property injection

A technique in which a set member function receives a pointer or reference to another object and stores it in the object on which the function is called to modify that object’s capabilities.

pseudocode

An informal language that helps you develop algorithms without worrying about the strict details of C++ language syntax.

pseudorandom

The `default_random_engine` actually generates these kinds of numbers.

pure virtual functions

A class is made abstract by declaring at least one of these.

put pointer

Indicates the byte number in the file at which the next output should be placed.

Q**quadratic run time**

An $O(n^2)$ algorithm is said to have this.

quantifier

In a regular expression, this specifies how many occurrences to match of the preceding subexpression.

quantum computers

Now under development. Theoretically, in one second, could do staggeringly more calculations than the total that have been done by all computing devices since the beginning of time.

R**RAII—Resource Acquisition Is Initialization**

For any resource that must be returned to the system when the program is done using it, the program should create the object as a local variable in a function. The object’s constructor should acquire the resource while initializing the object. The program should then use that object as necessary. When the function call terminates, the object goes out of scope. At this point, the object’s destructor should release the resource.

random_device

The proper way to randomize is to seed the random-number generator engine with this type of object (from header `<random>`).

raw string literals

A string literal in which the compiler automatically inserts backslashes as necessary to escape special characters like double quotes ("), backslashes (\), etc.

recursive function

A function that calls itself, either directly or indirectly (through another function).

reference parameter

An alias for its corresponding argument in a function call.

regular expression

A string that describes a search pattern for matching characters in other strings.

relational and equality operators

Operators `>`, `>=`, `<`, `<=`, `==` and `!=`, which are used to form conditions that compare values.

relational database

Data is stored in simple tables of rows and columns. Includes records and fields.

remainder operator, %

Also called the modulus operator, this yields the remainder after integer division and can be used only with integer operands.

representational error

Due to this, `10 / 3` can't be represented exactly in a fixed amount of memory.

resize

This `string` member function increases a `string`'s size by a specified number of characters.

rethrowing the exception

Doing so notifies the caller that the exception is not yet handled.

return 0;

When this appears at the end of `main`, the value 0 indicates that the program terminated successfully.

right brace, }

Ends each function's body, which contains the instructions the function performs.

Rule of Five

The C++ Core Guidelines state that if a class requires one special member function, it should define them all.

Rule of Five defaults

For classes with the compiler-generated special member functions, some experts recommend declaring them explicitly in the class definition with `= default`.

Rule of Zero

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.

rules of operator precedence

Specify the order in which C++ applies operators in expressions.

runtime polymorphism

Enables you to conveniently "program in the general" rather than "program in the specific."

rvalue references

Typically, these refer to temporary objects or objects about to be destroyed.

rvalues

Literals are said to be these because in an assignment, the can be used on only assignment operator's right side.

S**scientific notation**

This format is useful for displaying very large or very small values (that is, values with negative exponents).

scope

Defines a variable's lifetime and where it can be used in a program.

scope

The portion of a program where a particular identifier can be used.

scoped enumeration

This kind of user-defined type is introduced by the keywords `enum class`, followed by a type name and a set of identifiers representing integer constants.

scraping

Extracting data from text. Most commonly used to describe extracting data from websites.

searching

Determine whether a value is present in the data.

search key

Search algorithms determine whether this value is present in the data.

secondary storage devices

Computers store files on these—examples include flash drives and, frequently today, the cloud.

selection sort

An easy-to-implement but inefficient sorting algorithm. The algorithm's first iteration selects the smallest element value and swaps it with the first element's value. The second iteration selects the second-smallest element value (the smallest remaining one) and swaps it with the second element's value. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element's value, leaving the largest value in the last element.

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.

self-assignment

For `string myString`, the statement `myString = myString;` is an example of this.

semicolon. ;

Most C++ statements end with this. Omitting it when one is needed is a syntax error.

sentinel value

When entering and processing a series of values, this indicates end of data entry.

sequence of bytes

C++ views each file simply as this.

sequence structure, selection structure and iteration structure

Böhm and Jacopini's work demonstrated that all programs could be written in terms of only these control structures

shallow copy

If a data member is a pointer, this copies the pointer, not what it points to.

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`.

single-entry/single-exit control statements

There's only one way to enter and only one way to exit each control statement.

single inheritance

A derived class that inherits from one base class uses this kind of inheritance.

sizeof

This compile-time unary operator determines the size in bytes of a built-in array, type, variable or constant during program compilation.

size of a string

The number of characters stored in a string.

slicing

This logic error occurs when a derived-class object is copied into or assigned to a base-class object.

software

The C++ instructions you write, which also are called code.

software reuse

Using the existing C++ standard library and various open-source libraries to leverage your program-development efforts and avoid "reinventing the wheel."

sorting

Places data in ascending or descending order.

sort keys

Sorting places data in ascending or descending order based on one or more of these.

spaceship operator

C++20's new three-way comparison operator (`<=>`).

span (from header)

Enables programs to view contiguous elements of a container, such as a built-in array, a `std::array` or a `std::vector`.

special member functions

The default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator and destructor are known as these.

special member functions

A class's a copy constructor, copy assignment operator, move constructor, move assignment operator and destructor.

stack unwinding

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.

standard input stream object cin

Programs use this to obtain values from the keyboard.

standard library function pow

C++ does not include an exponentiation operator, so we use this instead from the header `<cmath>`.

standard library function terminate

Calls `abort` to terminate the program.

standard library swap function

Exchanges the contents of two objects.

starts_with member function of class string

Returns true if a string begins with a specified substring; otherwise, returns

static

Unlike other local variables, which exist only until a function call terminates, a local variable of this kind retains its value between function calls.

static binding

The compiler resolves at compile time the function to call.

static data member

Such a variable represents “class-wide” data shared by all objects of the class.

std:::

Specifies that we are using a name from that belongs to the C++ standard library's std namespace, such as cout.

std::cout

The standard output stream object, which is normally “connected” to the screen.

std::ranges::count_if algorithm

Counts all the elements in its first argument for which the lambda in its second argument returns true.

std::to_array function

This C++20 function converts a built-in array to a std::array.

stream extraction operator, >>

Used with the standard input stream object cin to obtain a value from the keyboard.

streams of data

Typically, C++ accomplishes output and input with these.

string

Consists of quotation marks ("") and the characters between them. Also known as a character string or string literal.

string_view

A read-only view of the contents of a C-string, a string object or a range of characters in a container, such as an array or vector of chars.

string class

Objects of this class store strings.

string concatenation

“Adding” string objects using the + operator.

string member function empty

Returns true if the string is empty; otherwise, it returns false.

string member functions find and rfind

Search from the beginning or end of a string, respectively.

string member function substr

Returns a string containing a portion of the string object on which the function is called.

string-object literal

Indicated by placing the letter s immediately after the closing " of a string literal.

string's overloaded += operator

For string concatenation assignment.

strong exception guarantee

If an exception occurs, objects' states remain unmodified or, if objects were modified, they're returned to their original states before the operation that caused the exception.

structured programming

Became almost synonymous with “goto elimination.”

subspan member function of a span

Returns a span representing a subset of the elements in a span.

supercomputers

Already can perform over one million trillion (a quintillion) instructions per second.

swap member function

This string member function exchanges the contents of two strings.

switch multiple-selection statement

Chooses among many different actions based on the possible values of a variable or expression.

switch selection statement

Performs one of many different actions (or groups of actions), depending on the value of an expression.

syntax error

Such as when one brace in a block is left out of the program. Has its effect at compile time.

T**terabyte**

Approximately one trillion bytes.

termination model of exception handling

Program control cannot return to the throw point.

this

A pointer that each object's member functions use to access the object.

three-way comparison operator (<=>)

For most types, the compiler can handle the comparison operators for you via the compiler-generated default implementation of this.

throw point

The location in the program from which an exception is thrown.

throw statement

Used to indicate to a caller that an exception occurred.

throw statement

Executed in a `catch` handler to return the current exception to the next enclosing `try` block.

tightly coupled

Describes the kind of closely related classes you create with implementation inheritance. Changes to a base class directly affect all corresponding derived classes.

`to_string` function

Returns the `string` representation of its numeric argument. This function is overloaded for the fundamental numeric types.

top

A single pseudocode statement that conveys the overall function of the program.

top-down, stepwise refinement

A process which is essential to the development of well-structured programs.

truncated

This is what happens to existing files opened with mode `ios::out`—all data in the file is discarded without warning.

type-safe linkage

Ensures that the proper function is called and that the types of the arguments conform to the types of the parameters.

U**UML class diagram**

Illustrates the relationships among classes in a hierarchy.

unary scope resolution operator, `::`

C++ provides this to access a global variable when a local variable of the same name is in scope.

undefined behavior

According to the C++ standard, this is the result of division by zero in floating-point arithmetic.

user-defined type

A custom class.

using declaration

Eliminates the need to repeat the `std::` prefix when using a specific standard library component such as `cout` or `cin`.

using directive

Enables your program, for example, to use names from the `std` namespace without the `std::` qualification.

utility function (also called a helper function)

A private member function that supports the operation of a class's other member functions and is not intended for use by the class's clients.

V**variable**

Location in the computer's memory where a value can be stored for use by a program.

virtual destructor

Every class that contains virtual functions should have one of these special

virtual functions

With these functions, the type of the object pointed to (or referenced)—not the type of the pointer (or reference)—determines which member function to invoke.

`void*`

This type can represent any pointer type.

volatile

This describes the information in the memory unit, which is typically lost when the computer's power is turned off.

W**what member function of an exception object**

Called on the exception to get its error message.

whitespace

Blank lines, spaces and tabs that make programs easier to read. These are normally ignored by the compiler.

Windows operating system

A proprietary operating system controlled exclusively by Microsoft. It is the world's most widely used desktop and notebook operating system..

workflow

A UML activity diagram models this.

X***xvalue***

An expiring value.

Z**zero-overhead principle**

You do not pay a price for a given feature unless you use it.

“

“administrative” section of a computer

This is the **Central Processing Unit (CPU)**, which coordinates and supervises the operation of the other logical sections.

“broadcast” operation

Applies the same operation to every element of a data structure.

“free” functions

These are not part of a class.

“manufacturing” section of a computer

This is the **Arithmetic and Logic Unit (ALU)**, which performs calculations and makes decisions.

“receiving” section of a computer

This is the **input unit**, which obtains information (data and computer programs) from input devices and places it at the other units' disposal for processing.

“shipping” section of a computer

This is the **output unit**, which takes the information the computer has processed and places it on various output devices to make it available outside the computer.

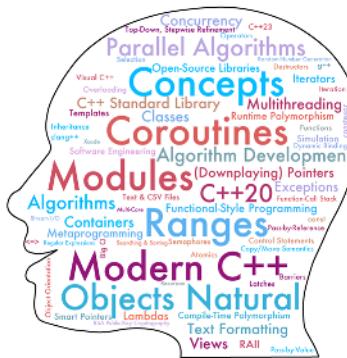
“warehouse” section of a computer

This is the rapid-access, relatively low-capacity **memory unit** that retains information entered through the input unit, making it immediately available for processing when needed.

“warehousing” section of a computer

This is the **secondary storage unit**, which provides long-term, high-capacity storage.

Index



Symbols

1076
 : in inheritance **539**
 :: (scope resolution operator) **244, 451**, 488
 unary 1181
 ! (logical negation) 169, **171**
 !, logical negation 1185
 != (inequality operator) 79, 1185
 ? (quantifier in regex) 403
 ?: (ternary conditional operator) **104**
 . (member selection operator) 456
 ' (digit separator) **207**
 '\0' (null character) **340**
 '\n' (newline character) 340
 "The Twelve Days of Christmas" Song exercise 187
 [[assume]] attribute 1193
 [[nodiscard]] attribute **1188**
 [] (operator for map) 738
 [] (regex character class) **402**
 [&] (lambda introducer, capture by reference) **780**
 [=] (lambda introducer, capture by value) **780**
 {n,} (quantifier in regex) **404**
 {n, m} (quantifier in regex) **404**
 * (multiplication operator) 76
 * (pointer dereference or indirection operator) **314**, 315
 * (quantifier in regex) **402**

*= (multiplication assignment operator) 125
 / (division operator) 76
 /* */ (multiline comment) 67
 // (single-line comment) **67**
 /= (division assignment operator) 125
 \ (regex metacharacter) **401**
 \' (single-quote-character escape sequence) 69
 \" (double-quote-character escape sequence) 69
 \\ (backslash-character escape sequence) 69
 \a (alert escape sequence) 69
 \D (regex character class) **401**
 \d (regex character class) 401, **402**
 \n (newline escape sequence) 69
 \r (carriage-return escape sequence) 69
 \S (regex character class) **401**
 \s (regex character class) **401**
 \t (tab escape sequence) 69
 \W (regex character class) **401**
 \w (regex character class) **401**
 & (address operator) 313, 315, 1185
 & (to declare reference) 219
 && (logical AND operator) 1185
 && (logical AND operator) 170
 &= (bitwise AND assignment operator) 745, 1185
 % (remainder operator) 76
 %= (remainder assignment operator) 125
 ^ (bitwise exclusive OR operator) 1185
 ^ (regex metacharacter) **402**
 ^= (bitwise exclusive OR assignment operator) 745, 1185
 + (addition operator) 74, 76
 + (quantifier in regex) **403**
 -- (postfix decrement operator) **125**
 ++ (postfix increment operator) **125**
 on an iterator 710
 -- (prefix decrement operator) **125**
 ++ (prefix increment operator) **125**
 on an iterator 710
 += (addition assignment operator) **125**
 string concatenation 361
 < (less-than operator) 79
 << (stream insertion operator) **68, 74**
 <= (less-than-or-equal-to operator) 79
 <=> (three-way comparison operator) 590, 636, **637**, 708
 <stacktrace> header (C++23) 1193
 = (assignment operator) 74, 76, 470, 596, 710
 -= (subtraction assignment operator) 125
 = 0 (pure specifier for a pure virtual function) **559**

- == (equality operator) 79
- > (arrow member selection) **456**
- > (greater-than operator) 79
- > (in a compound C++20 concept requirement) 876
- >= (greater-than-or-equal-to operator) 79
- >> (stream extraction operator) **74**
- | (bitwise inclusive OR operator) 1185
- | (operator in a C++20 range pipeline) **288**
- |= (bitwise inclusive OR assignment operator) 745, 1185
- || (logical OR operator) 169, **170**, 1185
- ~ (bitwise complement operator) 1185

- A**
- A (or a) presentation type **1129, 1130**
- abbreviated function template (C++20) 854, **854**, 870, 871, 882
 - constrained auto 870
- Abbreviated Function Template exercise 929
- abort standard library function **464**, 668, 679
- absolute value 192
- abstract class **558**, 559, 560, 571
 - Employee 560
 - pure **559**
- accelerometer 5
- access a global variable 244
- access function **456**
- access non-**static** class data
 - members and member functions 491
- access privileges 325, 327
- access shared data 1015
- access specifier **437**, 437, 480
 - private** 437
 - public** 437
- access the caller's data 219
- access violation 700
- Account Inheritance Hierarchy exercise 583
- accounts-receivable system 378
- accumulate algorithm **284**, 813, 822, 825, 838
- accumulator **349**, 350, 354
- accumulator overflow **354**
- ACM Data Science Task Force xxxix
- acquire
 - a lock **1019**
 - a semaphore 1059
- acquire member function of **std::binary_semaphore** **1061**, 1062
- action 106
- action expression in the UML **97**
- action state in the UML **97**, 176
 - symbol **97**
- action to execute **95**
- activation record **214**
- active block 1182
- activity diagram in the UML **97**, 106, 177
- activity in the UML **97**
- ad-hoc constraint (C++20 concepts) **876**
- adapter 739
- add an integer to a pointer 331
- adding strings **86**
- addition 6, 76, 77
 - compound assignment operator, **+=** **125**
- address operator (&) 313, 315, 316, 596
- adjacent_difference algorithm 838
- adjacent_find algorithm 837
- ADL (argument-dependent lookup) 857
- “administrative” section of the computer 6
- advance function **872**
- aggregate initialization 893
- aggregate type **492**, 517, 890
 - designated initializer 493
- agile software development **48**
- AI (artificial intelligence) xliv
- AI and Data Science case study exercise xxxi
- AI/Data Science Case Study exercises
- Machine Learning with Simple Linear Regression: Statistics Can Be Deceiving 417
- AI/Data Science case study exercises xxxi, 423, 424
- aiming a derived-class pointer at a base-class object 549
- air-traffic-control systems xxiii, 18
- alert escape sequence ('\a') 69
- algebraic expression 77
- algorithm **95**, 114, 237
 - development xxix, xl, xlii
- <algorithm> header 202, **618**, 627, 692, 720, 774, 837
- algorithm-development case studies xxix
- algorithms
 - binary search **1203**
 - bubble sort 1221, 1222
 - bucket sort 1222
 - insertion sort **1209**, 1210
 - merge sort **1213**
 - parallel xxxv
 - quicksort 1223
 - recursive binary search 1223

- algorithms (cont.)
 - recursive linear search 1223
 - rotate** 498
 - selection sort **1211**
- algorithms (standard library) **699**, 714
 - accumulate** **284**, 813, 822
 - binary_search** **277**
 - copy_backward** **803**
 - for manipulating containers 191
 - for_each** **631**
 - gcd** 813, **813**
 - iota** 813, **814**
 - is_sorted** **692**
 - iter_swap** 800, **801**
 - lcm** 813, **814**
 - max** **445**, 812, **812**
 - min** 812, **812**
 - minmax** 812, **812**
 - multipass **712**
 - partial_sum** 813, **815**
 - reduce** 813, **815**
 - separated from container 776
 - sort** **277**, 991
 - specialized memory 838
 - swap** 800, **800**
 - swap_ranges** 800
- alias 220
 - declaration (**using**) **899**, **1157**
 - for a type **899**, **900**, **1157**
 - for the name of an object 468
- aligned field 1118
- alignment 1130
- all**
 - algorithm 745
 - range adaptor (C++20) 829
- all_of** algorithm 837
 - ranges version (C++20) 796, **798**
- allocate memory 201, **598**, 598, 600
- allocator_type** 707
- AlphaGo 56
- alphanumeric character **401**
- AlphaZero **56**
- ALU (arithmetic and logic unit) **6**
- Amazon Alexa xlv, 63
- Amazon Web Services (AWS) 42
- ambiguity problem 1171, 1173
- American National Standards Institute (ANSI) 18
- Analyzing the Game of Craps exercise 752
- and** operator keyword **1185**
- and_eq** operator keyword **1185**
- Android
 - operating system **17**
 - smartphone 17
- angle brackets (< and >) 227
 - in templates **850**
- anonymous function 286, 778
- Anscombe's Quartet 418, 420
- Anscombe's Quartet exercise xxxi
- ANSI (American National Standards Institute) 18
- any algorithm 745
- <any> header 202
- any_of** algorithm **799**, 837
 - ranges version (C++20) 796, **799**
- Apache Software Foundation **15**
- APIs 43, 44
- append** **361**
- append data to a file 380
- Apple 16
 - Macintosh 16
 - Siri xlv, 63
 - TV **16**
 - Watch **16**
 - Xcode xxviii, liv
- Apple Code xxxvii
- application programming
 - case studies xlvi
- AR (augmented reality) **45**
- arbitrary precision integers
 - cpp_int** class from the Boost Multiprecision library **130**
- archive in JSON serialization with **cereal** 517
- argument coercion **198**
- argument-dependent lookup (ADL) 857
- arguments in correct order 195
- arguments passed to member-object constructors 474
- arithmetic
 - compound assignment operators **125**
 - function object **822**
 - operations 349
 - operator xxviii, **76**
 - overflow **681**
 - underflow **681**
- arithmetic and logic unit (ALU) **6**
- arithmetic mean 77
- arithmetic overflow **110**
- "arity" of an operator 597
- ARPANET 41
- array
 - built-in 320
 - C style 310
 - pointer based 310
- array** class template **260**, 670, 704
 - bounds checking 263, 264, **264**
 - container xxx
 - multidimensional **279**
- <**array**> header 201, 262, **278**
 - to_array** function (C++20) 311, **324**
- array names decay to pointers **322**

array subscript operator ([])
 610
 arrow member selection operator (->) 456, 483
 artificial general intelligence
 54, 62
 artificial general intelligence (AGI) 55
 artificial intelligence xxxi
 artificial intelligence (AI)
 xxviii, 54
 heuristic programming
 xxx
as_const function **896**
 ASCII (American Standard Code for Information Interchange)
 Character Set 90
 character set **10**
 ASCII (American Standard Code for Information Interchange) character set
 340
 assembler **13**
 assembly language **13**
assert
 contract keyword **688**
 macro **671**, 686
 macro to disable assertions
 671
assertion **671**
assign
 addresses of base-class and derived-class objects to base-class and derived-class pointers 547
 class objects 471
 one iterator to another
 713
assign member function
 list **727**
 string **360**
 assignment operator **125**,
 470, 596, 710
 *= 125
 /= 125
 %= 125
 assignment operator (cont.)
 += **125**
 -= 125
 = **74**
 default **470**
 assignment statement 74
 assisting people with disabilities 53
 associative container 712,
 713, 730, 732
 insert function 731, 735
 map 730
 multimap 730
 multiset 730
 ordered 704, **705**, 705,
 730
 set 730
 unordered 704, 706, 708,
 730
 unordered_map 730
 unordered_multimap
 730
 unordered_multiset
 730
 unordered_set 730
 associative container member functions
 contains **732**
 count **731**, 737
 equal_range **733**
 extract member function 709
 find **732**
 insert **733**, 737
 lower_bound **733**
 merge member function
 730
 upper_bound **733**
 associativity of operators **77**
 asterisk (*) **76**
 asynchronous
 concurrent threads 1014
 event **669**
 task 1076, 1089
 task completes 1074
 at member function 721
 array **264**, 686
 string **360**, 595
 vector **293**, 686
 Atomic Counters exercise
 1070
 <atomic> header **1048**
 atomic operation **1016**
 atomic pointer 1051
 atomic types **1048**
 std::atomic class template
 1049, 1049
 std::atomic_ref class template (C++20)
 1049, 1049, 1052
 thread safety 989
 atomic_ref class template (C++20) **1049**, 1049,
 1052
 attribute
 [[nodiscard]] **1188**
 [[fallthrough]] **163**
 of an object 23, 24
 audit
 contract level **688**
 level precondition 693
 augmented reality (AR) **45**,
 46
 auto keyword **281**, 718
 automated
 closed captioning 53
 automatic local variable 1182
 automatic storage class **1182**
 automatic storage duration
 1182, 1183
 automatically destroyed 241
 automobile as an object 23
 average 77
 average calculation 107, 109,
 114
 avoid
 naming conflicts 482
 repeating code 462
 await_ready function of an awaitable object (coroutines) **1095**

await_resume function of an awaitable object (coroutines) **1095**
await_suspend function of an awaitable object (coroutines) **1095**
 awaitable object (coroutines) **1089**, 1095
 await_ready function **1095**
 await_resume function **1095**
 await_suspend function **1095**
 awesome-public-datasets 414
 axiom contract level **688**

B

back member function
 queue **742**
 sequence containers **721**
 span class template
 (C++20) **338**
 vector **396**
 back_inserter function template **788**, 806, 807
 background_executor (concurrency) **1085**
 backslash
 \ 69
 escape sequence, \\ 69
 bad data **393**, 394
 bad member function 1126
 bad_alloc exception **598**, **676**, 681
 bad_cast exception **681**
 bad_typeid exception **681**
 bad_weak_ptr exception **1148**
 badbit of stream **1124**, 1126
 bandwidth 41
 Banker's rounding **182**
 banking systems xxiii, 18
 bar chart 185, 272
 printing program 273
 bar of asterisks 272

Bar-Chart Printing Program
 exercise 185
 barcode 5
 barrier (C++20) 1052, **1055**
 <barrier> header (C++20)
 203, **1055**
 Barriers exercie 1070
 base case **230**, 235, 236
 base class **530**, 534
 catch 682
 default constructor 544
 exception 681
 initializer **539**, 539
 pointer to a base-class object 547
 pointer to a derived-class object 547
 private member 1160
 subobject **1174**
 base e 192
 base-10 number system 192, 1111
 base-16 number system 1111
 base-8 number system 1111
 basic descriptive statistics 751
 Basic Descriptive Statistics exercise 751
 basic exception safety guarantee **664**
 BASIC programming language 21
 basic searching and sorting algorithms of the standard library 796
 basic_ios class 1174
 basic_iostream class **1104**, 1174
 basic_istream class **1103**, 1174
 basic_ostream class 1174
 begin
 function **278**, 323
 header <array> **278**
 member function of containers 708
 member function of first-class containers **710**
 beginning
 of a file 383
 of a stream 383
 behavior
 of a class 23
 bell 69
 Bell Laboratories 18
 bidirectional iterator 712, 723, 731, 734, 736, 884
 operations 714
 bidirectional_iterator concept (C++20) 777, 804
 bidirectional_range concept (C++20) 777, 803, 804, 806, 807
 big data xxiv, xxviii, **12**, 52, 358, 390
 Big Four C++20 features xxiv, xxxiv, 848
 Big O 1213, 1220
 binary search $O(\log n)$
 1219
 bubble sort, $O(n^2)$ 1219
 insertion sort, $O(n^2)$ 1219
 linear search, $O(n)$ 1219
 merge sort, $O(n \log n)$
 1219
 notation xxxvi, **700**, 702, **1198**, **1199**, 1207, 1210, 1213, 1219
 quicksort, best case $O(n \log n)$ 1219
 quicksort, worst case $O(n^2)$ 1219
 recursive binary search $O(\log n)$ 1219
 recursive linear search, $O(n)$ 1219
 selection sort, $O(n^2)$ 1219
 binary (base 2) format 1130
 binary digit (bit) 10
 binary function 823
 object **823**
 binary integer 140
 binary left fold **905**, 906, 909
 binary number system 10
 binary operator **74**, 76

- binary predicate function 725, 785, 797, 803, 807, 810
- binary right fold **906**, 909
- binary search 28, 691, 1198, **1203**, 1207, 1219
 - algorithm 701, 1207
 - binary_search** standard library algorithm **277**, **278**, 837
 - binary_search** standard library algorithm ranges version (C++20) 796, **798**
- binary search efficiency 1207
- binary tree **699**
- binary_semaphore**(C++20) **1059**
- <bit> header 203
- bit ("binary digit") **6**, **10**
- bit manipulation 744
- bitand operator keyword **1185**
- Bitcoin 9, 47, 51, 63, 129, 507, 1041
- bitor operator keyword **1185**
- bitset** 706, 744
- <biset> header 201
- bitwise
 - assignment operators 745
 - left-shift operator (<<) 588
 - operators xli
 - right-shift operator (>>) 588
- bitwise AND assignment **1185**
- bitwise assignment operator keywords 1185
- bitwise complement 1185
- bitwise exclusive OR 1185
- bitwise inclusive OR 1185
- bitwise operator keywords **1185**
- blank line 114
- block **82**, 104, 239, 241, 1182, 1183
 - is active 1182
 - is exited 1182
 - of memory 728
 - scope 239, **239**
 - thread until a lock is released **1019**
- blockchain 9, **46**, 51, 63, 129, 1041
- Blocked State discussion exercise 1069
- blocked thread state **1001**
- body
 - function **68**, 68
 - if** statement 80
 - of a loop 106
- Bohm, C. 96
- bool**
 - contextual conversion **613**
 - data type **100**
- bool** cast operator of a stream 1106
- boolalpha** stream manipulator **86**, 1111, 1122
- Boolean 100
- Boolean values in JSON 516
- Boost** **16**
 - Math library **418**, 421
- Boost C++ libraries xxvi, 19, 130
 - Boost Multiprecision library 20
- Boost Math library
 - simple_ordinary_least_squares** function **421**
- Boost Multiprecision Arithmetic Operations exercise 142
- Boost Multiprecision library 232
 - cpp_dec_float_50** class xxix
 - cpp_int** class xxix
- Boost Multiprecision open source library **130**
- Boost Multiprecision open-source library 157, 158, 180
 - cpp_dec_float_50** 180
- Boost.Log logging library 685
- born thread state **1000**
- Bounded Buffer
 - A Real-World Example discussion exercise 1069
- bounds checking **264**
- braced 618
- braced initializer **72**, 599
 - list 618
 - list as constructor argument 618
- list for custom classes 618
- narrowing conversion 198
- braced initializers as constructor arguments 618
- braces {} 68, 82, 104
 - not required 163
- bracket [] 261
- brain mapping 53
- branch 761
 - instructions 353
 - negative 761
 - zero 761, 764, 766, 768
- break** statement **163**, 167, 187
- brittle
 - base-class problem 1161
 - software **1161**
- broadcast operations 613
- brute force computing 55
- bubble sort 1219, **1221**, 1222
 - improving performance 1222
- bucket 1222
- bucket sort 1222, **1222**
- buffer** **1009**
- buffer is filled 1104
- buffer is flushed 1104
- buffer overflow 264
- buffered output **1104**
- buffered standard error stream 1103
- buffering 1127
- build level (contracts) **693**

building block appearance 177
 building-block approach 20
 building your own compiler
 case study xxx, xxxiv, xliv,
 13, 351, 753, 757, 759,
 760, 763, 765, 766, 767,
 769, 771
 built-in array xxx, 310, 320
 byte 6, 10

C

C xxxv, xlvi
 .C extension 25
 C-like pointer-based array
 706
 c presentation type 409,
 1130, 1130
 C programming language 18
 C-string 310, 340
 C-style arrays 310
 C-style string 310
 C# programming language 21
 C++
 code repositories (on
 GitHub) xxxix
 Language Reference xlvi
 open-source community
 xxxix
 C++ Core Guidelines xxvii,
 xli, 19, 972
 explicit single-parameter
 construtor 440
 Guidelines Support Li-
 brary 199
 override 556
 C++ documentation xlvi
 C++ *How to Program* xlvii
 instructor resources xlv
 C++ language documentation
 (Microsoft) xlvi
 C++ preprocessor 67
 C++ programming language
 C++ standard library 19
 code repositories (on
 GitHub) 20
 compiler 26
 preprocessor 26

C++ standard
 document xlii
 C++ standard library xxvi, 19,
 66, 191
 array class template 260
 container 260
 exception types 681
 headers 200
 string class 83, 436
 <string> header 85
 vector class template 290
 C++11 xxiv, 19
 C++14 xxiv, 19
 C++17 xxiv, 19
 C++20 xxiv, 19
 abbreviated function tem-
 plate 854, 854
 ad-hoc constraint in con-
 cepts 876
 <barrier> header 1055
 bidirectional_ite-
 rator concept 804
 bidirectional_range
 concept 803, 804, 806,
 807
 “big four” features xxiv,
 xxxiv, 848
 C++ standard document
 xlvi
 co_await operator
 (coroutines) 1074
 co_return statement
 (coroutines) 1074,
 1088
 co_yield expression
 (coroutines) 1074,
 1077, 1079
 <compare> header 637
 concept keyword 868
 concepts xxvii, xxxiv, 774,
 776, 856, 860, 872
 concepts by header 862
 <concepts> header 861
 conjunction in a con-
 straint or concept 862
 consteval function 919
 constrained auto 870

C++20 (cont.)
 constraint expression in a
 concept 860, 868
 constraint in concepts
 860, 861
 contiguous_iterator
 concept 782
 contracts (pushed to a later
 standard) 687
 coroutine 1074
 coroutines xxxv
 disjunction in a constraint
 or concept 862
 ends_with member func-
 tion of class string 86
 forward_iterator con-
 cept 787, 794
 forward_range concept
 787, 789, 794, 798,
 804, 810, 811
 indirectly_copyable
 concept 779
 indirectly_readable
 concept 779
 indirectly_swappable
 concept 801
 indirectly_writable
 concept 779, 790
 input_iterator concept
 788, 805
 input_or_output_it-
 erator concept 783
 input_range concept
 779, 784, 785, 786,
 788, 790, 791, 792,
 793, 795, 796, 797,
 798, 799, 801, 803,
 804, 805, 806, 808,
 809, 813
 iterator concepts 777
 <latch> header 1052
 modules xxvii, xxxv
 output_iterator con-
 cept 782
 output_range concept
 782
 permutable concept 793

- C++20 (cont.)
- positional argument in a format string 1129
 - projection in a ranges algorithm 785
 - projection in `std::ranges` algorithms 825
 - `random_access_iterator` concept 793, 797, 818
 - `random_access_range` concept 793, 797
 - range 286, 699
 - range adaptor 829
 - range concepts 777
 - range-based algorithm xxxiv
 - ranges xxx, xxxiv
 - `<ranges>` header 286
 - ranges library 286, 393
 - requires clause 860
 - requires expression 874
 - `<semaphore>` header 1059
 - sentinel of a range 722
 - standard concepts by header 862
 - `std::all_of` algorithm (ranges) 796, 798
 - `std::any_of` algorithm (ranges) 796, 799
 - `std::atomic_ref` class template 1049, 1049, 1052
 - `std::barrier` 1052, 1055
 - `std::binary_search` algorithm (ranges) 796, 798
 - `std::binary_semaphore` 1059
 - `std::copy` algorithm (ranges) 722, 778
 - `std::copy_backward` algorithm (ranges) 802, 803
 - `std::copy_if` algorithm (ranges) 802, 804
 - `std::copy_n` algorithm (ranges) 802, 805
 - `std::count` algorithm (ranges) 792, 793
 - `std::count_if` algorithm (ranges) 792, 793
 - `std::counting_semaphore` 1059
 - `std::equal` algorithm (ranges) 783, 784
 - `std::equal_range` algorithm (ranges) 810
 - `std::fill` algorithm (ranges) 781, 782
 - `std::fill_n` algorithm (ranges) 781, 782
 - `std::find` algorithm (ranges) 796, 796
 - `std::find_if` algorithm (ranges) 796, 797
 - `std::find_if_not` algorithm (ranges) 796, 799
 - `std::for_each` algorithm (ranges) 779
 - `std::format` function from header `<format>` 152, 155, 1127
 - `std::generate` algorithm (ranges) 781, 782
 - `std::generate_n` algorithm (ranges) 781, 783
 - `std::includes` algorithm (ranges) 807, 808
 - `std::inplace_merge` algorithm (ranges) 805, 806
 - `std::jthread` 1003, 1003, 1008
 - `std::latch` 1052, 1052, 1053
 - `std::lexicographical_compare` algorithm (ranges) 783, 786
- C++20 (cont.)
- `std::lower_bound` algorithm (ranges) 810, 810
 - `std::make_heap` algorithm (ranges) 818
 - `std::max_element` algorithm (ranges) 792, 794
 - `std::merge` algorithm (ranges) 802, 803
 - `std::min_element` algorithm (ranges) 792, 794
 - `std::minmax` algorithm (ranges) 813
 - `std::minmax_element` algorithm (ranges) 792, 794
 - `std::mismatch` algorithm (ranges) 783, 785
 - `std::move` algorithm (ranges) 803
 - `std::move_backward` algorithm (ranges) 803
 - `std::none_of` algorithm (ranges) 796, 799
 - `std::pop_heap` algorithm (ranges) 819
 - `std::push_heap` algorithm (ranges) 819
 - `std::ranges` namespace 722, 778, 779, 781, 783, 786, 790, 792, 796, 800, 802, 805, 807, 810, 812, 817
 - `std::remove` algorithm (ranges) 786, 787
 - `std::remove_copy` algorithm (ranges) 786, 788
 - `std::remove_copy_if` algorithm (ranges) 786, 790
 - `std::remove_if` algorithm (ranges) 786, 789
 - `std::replace` algorithm (ranges) 790, 790
 - `std::replace_copy` algorithm (ranges) 790, 791

- C++20 (cont.)
- `std::replace_copy_if` algorithm (ranges) 790, **792**
 - `std::replace_if` algorithm (ranges) 790, **791**
 - `std::reverse` algorithm (ranges) 802, **804**
 - `std::reverse_copy` algorithm (ranges) 805, **807**
 - `std::same_as` concept **869**
 - `std::set_difference` algorithm (ranges) 807, **808**
 - `std::set_intersection` algorithm (ranges) 807, **808**
 - `std::set_symmetric_difference` algorithm (ranges) 807, **809**
 - `std::set_union` algorithm (ranges) **809**
 - `std::shuffle` algorithm (ranges) 792, **793**
 - `std::sort` algorithm (ranges) 796, **797**, 826, 827, 828
 - `std::sort_heap` algorithm (ranges) **818**
 - `std::span` class template of header `` 311, **334**
 - `std::starts_with` member function of class `string` **86**
 - `std::stop_callback` for cooperative cancellation **1040**
 - `std::stop_source` for cooperative cancellation **1039**
 - `std::stop_token` for cooperative cancellation **1039**
- C++20 (cont.)
- `std::swap_ranges` algorithm (ranges) **801**, 801
 - `std::to_array` function of header `<array>` 311, **324**
 - `std::transform` algorithm (ranges) 792
 - `std::unique` algorithm (ranges) 802, **804**
 - `std::unique_copy` algorithm (ranges) 805, **806**
 - `std::upper_bound` algorithm (ranges) 810, **811**
 - `<stop_token>` header **1037**
 - templated lambda 856
 - text formatting xxix, 1127
 - three-way comparison operator (`<=>`) **637**, 708
 - `view` **286**, 699, 828
 - `viewable_range` **829**
 - `weakly_incrementable` concept 779
- C++20 modules
- transition from the pre-processor 938
- C++20 ranges
- `|` operator in a range pipeline **288**
 - pipeline 288
 - `rotate` algorithm 498
 - `std::views::filter` **288**
 - `std::views::iota` **287**
- C++20 spans xxx
- C++20 text formatting xxxvi
- format specifier 155
- C++23 xxiv
- `[[assume]]` 1193
 - `<stacktrace>` header 1193
 - concurrent map 1063
 - concurrent queue 1062
 - container enhancements 1192
- C++23 (cont.)
- contracts (could be later than C++23) 687
 - defect reports 1189
 - `generator` 1191
 - modular standard library 973
 - `print` function 1190
 - `println` function 1190
 - ranges 1190
 - `ranges::contains_-subrange` 1190
 - `ranges::ends_with` 1191
 - `ranges::find_last` 1190
 - `ranges::find_last_if` 1190
 - `ranges::find_last_if_not` 1191
 - `ranges::starts_with` 1191
 - `ranges::to` 1190
 - `std::mdarray` container 283
 - `std::mdspan` 1192
 - `views::enumerate` 1191
 - `views::zip` 1191
 - `views::zip_transform` 1191
- c++23
- ranges and views enhancements 1190
- C++23) 1189
- C++26 xxiv
- C++How to Program*
- code download liii
- Caesar cipher 246
- Caesar substitution cipher 494
- calculate a salesperson's earnings 137
- Calculating a Circle's Diameter, Circumference and Area exercise 141
- Calculating Number of Seconds exercise 251

- Calculating Sales exercise 185
 Calculating Students' Averages with Fold Expressions exercise 930
 Calculating the Product of Odd Integers exercise 184
 Calculating the Value of π exercise 186
 calculations 6, 97
 California Consumer Privacy Act (CCPA) xlv, 63
 callback function **1074**
 calling functions by reference 316
 camel case **73**
 cancer diagnosis 53
 capacity
 - of a `string` **365**
 - of a `vector` 715, **716**
 capacity member function
 - of `string` **365**
 - of `vector` **716**
 Capturing Substrings exercise 412
 capturing variables in a lambda **397**, 633
 Card Shuffling and Dealing 513, 514
 caret (^) regex metacharacter **402**
 carriage return ('\r') escape sequence 69
 cascading
 - member function calls **484**, 484, 486
 - stream insertion operations **74**
 case insensitive 406
 regular expression 401
case keyword **162**
 case sensitive **73**, 406
 regular expression 401
 case studies xxiv
 algorithm development xxix
 case studies (cont.)
 - An Intro to Similarity Detection with Very Basic Natural Language Processing, "Who Wrote the Works of William Shakespeare?" xxxi
 - Anscombe's Quartet xxxi
 - application programming xlivi
 - Building a Casino Dice Game xxix
 - Building Your Own Compiler xxx, xxxiv
 - Building Your Own Compiler exercise xxx
 - Building Your Own Computer xxxiv
 - Building Your Own Computer (as a virtual machine) xxx
 - Building Your Own Computer (as a virtual machine) exercise xxx
 - C++20 Text Formatting xxxvi
 - Card Shuffling and Dealing exercise xxxii
 - Contracts xxxiii
 - Coordinating Threads with Latches, Barriers and Semaphores xxxv
 - Counter-Controlled Iteration xxix
 - Creating a Coroutine with `co_await` and `co_return` xxxvi
 - Creating a Generator Coroutine xxxvi
 - Creating and Using Objects of Standard Library Class `string` xxviii
 - Cryptography with the Vigenère secret-key cipher xxix
 - case studies (cont.)
 - Implementing a `MyArray` Custom Class Template with Custom Iterators xxxv
 - Implementing RSA Public-Key Cryptography exercise xxxii
 - Implementing the `MyArray` Class with Overloaded Operators xxxiii
 - Intro to Similarity Detection with Very Basic Natural Language Processing exercise 424
 - Launching Tasks with `concurrentpp` xxxvi
 - Machine Learning with Simple Linear Regression exercise xxxi
 - Machine Learning with Simple Linear Regression: Statistics Can Be Deceiving exercise xxxi, 417
 - Machine Learning with Simple Linear Regression: Time Series Analysis exercise 423
 - multithreading and multicore systems performance xxxv, xxxvi
 - Nested Control Statements xxix
 - Precise Monetary Calculations with the Boost Multiprecision Library xxix
 - Producer–Consumer Synchronizing Access to Shared Mutable Data xxxv
 - Profiling Sequential and Parallel Sorting Algorithms xxxv
 - Random Number Simulation xxix

- case studies (cont.)
 - Runtime Polymorphism with an Employee Class Hierarchy xxxii
 - Sentinel-Controlled Iteration xxix
 - Studying the Vigenère Secret-Key Cipher Implementation xxxii
 - Super-Sized Integers with the Boost Multiprecision Library xxix
 - systems programming xlivi
 - The Tortoise and the Hare Race exercise xxx
 - Vizualizing the High-Speed Binary Search Algorithm xxxvi
 - Vizualizing the High-Speed Merge Sort Algorithm xxxvi
- casino 209
- <cassert> header 202, **671**
- cast **90**
 - cast operator 333, 640
 - cast away const-ness 890
 - overloaded **630**
 - catch block **295**
 - catch exceptions in constructors 673
 - catch handler 663, 668
 - all exceptions with catch(...) 682
 - base-class exception 682
 - catch related errors 682
 - catch(...) (catch all exceptions) 682
- Catching All Exceptions 696
- Catching Derived-Class Exceptions 696
- categorical data 414
- cbegin
 - member function of containers 708
 - member function of vector **718**
- CCPA (California Consumer Privacy Act) xlvi, 63
- <cctype> header 201
- ceil function 192
- Celsius and Fahrenheit Temperatures exercise 251
- cend
 - member function of containers 708
 - member function of vector **718**
- cend container member function **718**
- center-align text 1131
- central processing unit (CPU) **6**
- cereal header-only library 390, **516**
 - JSONInputArchive 520
 - JSONOutputArchive 519
- cerr (standard error stream) **28**, 377, 1103, 1104
- <cfloat> header 202
- chain of constructor calls **544**
- chain of destructor calls **544**
- chaining stream insertion operations **74**
- Challenge Project: The RSA Problem 527
- Challenging Research Project: Elliptic Curve Cryptography exercise 528
- char data type 72, 199
- char16_t 1103
- char32_t 1103
- char8_t 1103
- character **10**
 - set **10**
- character array 340
- character class (regular expressions) **401**, 402
 - custom **402**
- character constant **340**
- character literal 164, **164**
- character presentation 202
- character sequence 385, 436
- character set **90**
- character string **68**
- chatbots 54, **425**
- ChatGPT xlvi, 16, 57
- checkerboard pattern 90, 141
- Asterisks exercise 141
- Checkpoint exercises xlivi
- chess 55
- <chrono> header 201, 448, **993**
 - duration_cast **993**
 - steady_clock **993**
- chrono library 515
- cin (standard input stream) 27, 74, 377, 380, 1103, 1104
- cin.eof 1106
- cin.get function 1108
- cipher
 - Caesar 246
 - substitution 246, **246**
 - Vigenère 246, 248, 502, 505
- ciphertext **246**, 497, 522
- ciphertext in cryptography 497
- Circle Area exercise 256
- circular buffer **1027**
- Circular Buffer with Semaphores exercise 1070
- circular type reference 1148
- circular wait (necessary condition for deadlock) 1002
- circularly referential data **1148**
- clamp algorithm 838
- Clang C++ xxxvii, liv
 - clang++ in a Docker container 28
- clang++ compiler xxviii
- clang-tidy static analysis tools xxxvii, lvii
- class **19**, **23**
 - class keyword 227, 435, **435**, 850
 - constructor **438**
 - data member **24**
 - default constructor **441**

- class (cont.)
 development 604
 diagram in the UML **533**
 hierarchy **533**
 implementation programmer **454**
 interface **448**
 interface described by function prototypes **195**
 invariant **460**, 686
public services **448**
- Class Average exercise
 Creating a Grade Report from a CSV File 411
 Reading Grades from a Plain Text File 411
 Reading Student Records from a CSV File 411
 Writing Grades to a Plain Text File 411
- class-average problem 107, 108, 115
- class scope **239**, 451, 455
static class member 488
- class template **262**, 847, 849
 definition 849
 member-function templates 851
 scope 854
 specialization 849
 Stack 850, 852
- class template argument deduction (CTAD) **265**, 291, 719, 734, 786, 893, 896
- class template specialization **262**
- classes
array class template **260**
bitset 706, 744
Complex 645
deque 715, **728**
exception **681**
forward_list 715
HugeInt 647
invalid_argument 682
list 715, **723**
multimap 736
- classes (cont.)
MyArray 606
numeric_limits 110, 132
out_of_range exception class 295
Polynomial 651
priority_queue **743**, 816, 817, 818
queue **742**, 742
RationalNumber 651
runtime_error **658**, 668
set 734
shared_ptr **1143**
stack **740**
steady_clock **993**
string **83**, 436
system_clock **993**
tuple **899**
unique_ptr **601**, 1143
vector 290
- classic stream libraries **1103**
- cleaning data 400
- clear** function of **ios_base** **1126**
- clear** member function of containers 709, 723
- client
 of a class **442**
- client-code programmer **454**
- Climate at a Glance time series 423
- Climate at a Glance” time series 423
- <climits>** header 202
- CLion IDE (Jetbrains) 26
- clog** (standard error buffered) 377, 1103, 1104
- closed set of types **1154**
- cloud xxiv, 42, 515
 computing **42**
- cloud services
 Amazon Web Services (AWS) 42
 Google Cloud Platform (GCP) 42
 Microsoft Azure 42
- cloud-based services **42**, 515
- cmatch** **405**
- <cmath>** header 156, 192, 201
isnan function **395**
 list of functions 192, 193
- mathematical special functions **193**
- co_await** expression (C++20) **1089**
- co_await** operator (C++20) 1074
- co_return** statement (C++20) 1074, **1088**
- co_yield** expression (C++20) 1074, **1077**, 1079
- code **2**, **24**
 code download liii
 code repositories (on GitHub) xxxix, 20
- CodeLite 26
- coefficient 651
- Coffman, E. G. 1002
- coin tossing 204, 252
- Coin Tossing exercise 252
- collision in a hashtable **703**
- colon (:) 1171
 in inheritance **539**
- column **279**
- column headings 264
- combining Class Time and Class Date exercise 512
- combining control statements in two ways 173
- comma operator (,) 910
- comma-separated list 72, 82, 149
 of base classes 1171
 of parameters 195
- command-line argument **341**
- Command Prompt window 30
- comment **67**, 73
 multiline **67**
 single-line **67**
- CommissionEmployee** class header 565
 implementation file 566
 test program 537

- common programming errors xxvii
 common range 721, 775, 778
common range adaptor (C++20) 829
 communications systems xxiii, xliv, 18
CommunityMember class hierarchy 533
 commutative operators **635**
 comparator function object 730, 736
 less **730**, 743
<compare> header 203, 637
 compare iterators 714
 compare member function of class **string** **363**
 comparing **strings** 361
 compilation error **68**, 123
 also called a compile-time error **68**
 compilation phase 68
 compilation process 765
 compile **26**, 899
 with **clang++** 36
 with **g++** 33
 with Visual C++ 30
 compile a header as a header unit 940
 compile time
 calculations xxxv, 848
 compile-time constant 899
 polymorphism 710, **847**, 849
 predicate 860
 programs that write code 848
 recursion 902, 903
 static polymorphism **847**
 compiler **14**, 67, 68, 116
 Apple Xcode liv
 GNU **g++** liv, 3, 28, 32
 LLVM **clang++** 3, 28
 Microsoft Visual Studio liv, 28
 compiler (cont.)
 Visual Studio Community edition 3
 Xcode on macOS 28
 compiler error **68**
 Compiler Explorer website ([godbolt.org](https://www.godbolt.org)) **689**
 -pthread compiler flag 1003
 compiler optimization **767**
 compiler warnings
 enable 174
 compilers
 clang++ xxviii, xxxvii
 g++ xxviii, xxxvii
 Visual C++ xxviii, xxxvii, liv
 Xcode xxxvii
compl operator keyword **1185**
 completion function **1055**, 1058
 barriers **1055**
Complex class 510, 645
 exercise 510
Complex class member-function definitions 646
 complex numbers 510, 645
 complexity theory 237
 component 23
 composable **286**
 composable views **286**, 828
 composition **474**, 478, 530, 534
 as an alternative to inheritance 583
 exercise 511
 compound assignment operators **125**, 127
 compound interest 153, 185, 187
 compound requirement in C++20 concepts 874, **875** → 876
 compound statement **82**
 Compound-Interest Program (modified) exercise 185
 computational thinking xl
 computer-assisted instruction (CAI) 257, 258
 computer-assisted instruction (CAI): Difficulty Levels 258
 computer-assisted instruction (CAI): Monitoring Student Performance 258
 computer-assisted instruction (CAI): Reducing Student Fatigue 257
 computer-assisted instruction (CAI): Varying the Types of Problems 258
 computer dump **352**
 computer hardware xxiv
 computer science xxviii
 computer-science topics xxxvi
 computer simulator 351
 computer software xxiv
 computer vision 53, 63
 applications 54
 computers in education 257
Computing Competencies for Undergraduate Data Science Curricula Final Report xxxix
Computing Curricula 2020 xxxix
 computing the sum of the elements of an array 271, 284
 Computing Values at Compile-Time exercise 930
 concatenate 361
 concatenate stream insertion operations **74**
 concept-based overloading (C++20) **872**, 879, 913, 919
 Concept Constrained Abbreviated Function Template exercise 929
concept keyword (C++20) **868**

- concepts (C++20) xxxiv, 776, 856, 860, 872
 -> in a compound requirement 876
 ad-hoc constraint **876**
bidirectional_iterator 777, 804
bidirectional_range 777, 803, 804, 806, 807
 compound requirement 874, **875**
concept keyword **868**
conjunction **862**
constraint **860**, 861
constraint expression **860**, 868
contiguous_iterator 777, 782
contiguous_range 777
 custom 868
 disjunction **862**
forward_iterator 777, 787, 794
forward_range 777, 787, 789, 794, 798, 804, 810, 811
indirectly_copyable 779
indirectly_readable 779
indirectly_swappable 801
indirectly_writable 779, 790
input_iterator 777, 788, 805, 873
input_or_output_iterator 783
input_range 777, 779, 784, 785, 786, 788, 790, 791, 792, 793, 795, 796, 797, 798, 799, 801, 803, 804, 805, 806, 808, 809, 813
 iterators 777
 concepts (C++20) (cont.)
 listed by header **862**
 logical AND (**&&**) operator in a constraint 862
 logical OR (**||**) operator in a constraint 862
 nested requirement 874, **876**
 output_iterator 777, 782
 output_range 777, 782
 permutable 793
 random_access_iterator 777, 787, 788, 793, 797, 818, 873
 random_access_range 777, 793, 797, 818, 819
 ranges 777
 requires clause **860**
 requires expression **874**
 simple requirement 874, **874**
 standard 860
 std::floating_point **861**, 868
 std::integral **861**, 868
 std::same_as **869**
 type requirement 874, **875**
 weakly_incrementable 779
<concepts> header (C++20) 203, **861**
 concepts(C++20) xxvii
 concrete class **558**
 concrete derived class 562
concurrentpp coroutine support library **1076**
background_executor **1085**
 executor 1076
inline_executor **1085**, 1085
 install 1077
result **1081**
runtime **1081**, 1083
concurrentpp coroutine support library (cont.)
submit function of an executor **1084**
task 1076, **1081**
thread_executor **1084**
thread_pool_executor **1081**, 1084, **1084**
 timer 1076
 utility functions 1076
when_all function **1088**
when_any function **1089**
worker_thread_executor **1085**
concurrentpp::runtime
 discussion exercise 1098, 1099
concurrency
 vector hardware operations 991
concurrent container
 Google Concurrency Library (GCL) 1062
 Microsoft Parallel Patterns Library 1062
concurrent map (C++23) 1063
 reference implementation 1063
concurrent operations 988
concurrent programming 191, **989**
 with a simple sequential-like coding style 1074
concurrent queue (C++23) 1062
 reference implementations 1062
concurrent threads 1015
Condense Spaces to a Single Space exercise 412
condition **79**, 105, 158
 Yoda 174
condition variable 1021
condition_variable
 wait function **1021**

condition_variable class
1019
<condition_variable>
 header 202, **1019**
condition_variable_any
 class **1037**
conditional expression **104**
conditional operator, ?: 104
conditional transfer of control
 351
confusing equality (==) and
 assignment (=) operators
 173
conjunction in a C++20 con-
straint or concept **862**
conserving memory 1183
const 472
 keyword 207
 member function **436**, 472
 member function on a
 non-**const** object 473
 objects and member func-
 tions 473
 qualifier 270
 qualifier before type speci-
 fier in parameter decla-
 ration 221
 version of **operator[]** 629
const_cast
 cast away **const-ness** **890**
const_cast operator 1184
const_iterator 707, 708,
 713, 732
const_pointer 707
const_reference 707
const_reverse_iterator
 707, 708, 709, 713, 719
constant 754
 compile-time 899
constant integral expression
 164
constant pointer
 to an integer constant 327
 to constant data 325, 327,
 328
 to nonconstant data 325,
327
constant run time **1199**
constant running time **700**
constant variable 270
constexpr function
 (C++20) **919**
constexpr function **919**
constexpr if **919**
constexpr qualifier **270**,
 270
 “**const-ness**” 1184
constrained auto (C++20)
870
constraint **860**, **868**
constraint (C++20 concepts)
860, 861
constraint expression (C++20
 concepts) **860**, 868
constructor **438**, 441
 braced-initializer list **618**
 call chain 544
 conversion **640**, 642
 copy 620, 621
 default arguments 460
 deleted 1169
 exception handling 672
explicit 642
 function prototype 449
 in a class hierarchy 544
 inherit 1169
 inherit from base class
 1168
injection **576**
 multiple parameters 444
 single argument 642, 643
Constructors Throwing Ex-
ceptions 696
consume memory 238
consumer 989, **1008**
 thread **1009**
container 191, 201, **698**, 704
begin function 708
cbegin function 708
cend function 708
clear function 709
crbegin function 709
crend function 709
emplace function 709
container (cont.)
empty function 709
end function 708
erase function 709
insert function 709
map associative container
 730
map class template 705
max_size function 709
multimap associative con-
 tainer 730
multimap class template
 705
multiset associative con-
 tainer 730
multiset class template
 705
nested type names 892
priority_queue class
 template 706
queue class template 706
rbegin function 708
rend function 709
sequence 704, **704**, 704
set associative container
 730
set class template 705
size function 709
special member functions
 707
stack class template 706
swap function 709
unordered_map associa-
 tive container 730
unordered_multimap as-
 sociative container 730
unordered_multiset as-
 sociative container 730
unordered_set associa-
 tive container 730
container (Docker) xxxvii, **lv**
container adaptor 704, **706**,
 706, 713, **739**, 739
priority_queue **743**,
 816, 817, 818
queue **742**
stack **740**

- container adaptor functions
pop **739, 740**
push **739, 740**
- container in the C++ standard library **260**
- container member function
 complete list 707
- containerization **49**
- contains** function of associative container **732**
- contextual conversion **613**, 630, 644
- contextual conversion to **bool** **613**
- contiguous iterator 712, 776
- contiguous_iterator** concept (C++20) 777, 782
- contiguous_range** concept (C++20) 777
- continuation mode (for contract violations) **691**
- continue** statement **167**, 167, 187
- contract 685, 687
assert contract keyword **688**
 attributes 688
audit contract level **688**
axiom contract level **688**
build level **693**
 continuation mode **691**
contractViolation 693
default contract level **688**
 default violation handler **690**
 design by contract **687**
 disable contract checking 691
 early access implementations (GNU C++) 689
ensures contract keyword **688**, 689
expects contract keyword **688**, 689, **694**
- contract (cont.)
 experimental implementation xxxiii, 657
handleContractViolation default contract violation handler **690**
 level 688, 692, 693
post contract keyword (GNU C++ early access implementation) **689**
pre contract keyword (GNU C++ early access implementation) **689**
 proposal **688**
 violation **690**, 693
 violation handler **694**
- contractViolationObject** 693
- control statement xxix, **95**, 97, 99, 237
do...while 158
for 98, **148**, 148, 156
if **79**
 nesting **99**, 177
 stacking **99**, 176
switch **159**
while **106**, 147
- control variable **107**, 146, 147, 148
- controlling expression of a **switch** **162**
- converge on a base case 230
- conversion constructor 590, **640**, 642
- conversion operator 590, **630**, 640
explicit 643
- convert among user-defined types and built-in types 640
- convert between types 640
- convert lowercase letters 201
- convert **strings** to floating-point types 373
- convert **strings** to integral types 373
- Converting Integers to Characters exercise 409
- Converting Integers to Emojis exercise 410
- Cooking with Healthier Ingredients 416
- cooperative **1037**
- cooperative cancellation 1008, **1037**, 1039
std::stop_callback **1040**
std::stop_source **1039**
std::stop_token **1039**
- cooperative multitasking 1075
- cooperative thread cancellation 1037
- coordination types (thread synchronization) 1052
- coprime **523**
- copy 659
- copy algorithm 615, 618, **720**, 837
 ranges version (C++20) **722**, 778
- copy-and-swap idiom 621, 636
 strong exception guarantee 664
- copy assignment operator (=) xxxiii, 441, 589, **605**, 621, 710
 overloaded **592**
- copy-constructible type 659
- copy constructor xxxiii, 441, **472**, 478, 589, 593, 605, 608, 611, 620, 621, 708, 710
 default 478
- copy of the argument 325
- copy semantics xxxiii, 589, **605**
- copy_backward** algorithm **803**, 837
 ranges version (C++20) **802**, **803**

- copy_if** algorithm 837
 ranges version (C++20) 802, **804**
- copy_n** algorithm 837
 ranges version (C++20) 802, **805**
- CopyConstructible** 710
- coroutine** **1080**
 coroutine (C++20) 1074
 awaitable object **1095**
 co_yield expression **1077**, 1079
 coroutine frame **1095**
 coroutine state **1095**
 coroutine support library 1075, 1089
 coroutine_handle **1095**
 generator **1077**
 generator coroutine support library (Sy Brand) 1077
 promise object **1094**
 stackless **1080**
 suspend_always **1094**
 suspend_never **1094**
 suspension point **1095**
- <**coroutine**> header 203
- <**coroutine**> header (C++20) **1094**
- coroutine libraries
 concurrentpp **1076**
 cppcoro **1076**
 folly::coro **1076**
 generator (Sy Brand) **1076**, 1077
- coroutines (C++20) xxxv
- corpus** **424**
 corpora (plural of corpus) **424**
- correct number of arguments 195
- correct order of arguments 195
- cos** function 192
- cosine 192
- count** algorithm 837
 ranges version (C++20) 792, **793**
- count** function
 of associative container **731**
 of **multimap** 737
- count_statistic** **418**, 751
- count_down** member function of a **std::latch** **1053**
- count_if** algorithm 837
 ranges version (C++20) 792, **793**
- count_if** ranges algorithm (C++20) **396**, 397
- counted range adaptor** (C++20) 829
- counter** **107**, 113, 119
- counter-controlled iteration xxix, **107**, 108, 116, 119, 120, 146, 147, 148, 238
- Counting Characters and Words exercise 412
- counting loop 147
- counting word frequencies **426**
- counting_semaphore** (C++20) **1059**
- cout** (standard output stream) 27, 68, 71, 377, 1103, 1104
- .**cpp** extension 25
- .**cpp** filename extension 944
- cpp_dec_float_50** (Boost Multiprecision open-source library) 180
- cpp_int** class from the Boost Multiprecision library **130**
- cppcheck** static analysis tools xxvii, xxviii, lvii
- cppcoro** coroutines library **1076**
- cpplang Slack channel lvii
- .**cppm** filename extension 944
- CPU (central processing unit) **6**, 27, 991
- cracking RSA ciphertext 527
- Crafting Valuable Classes case study xxxiii, xxxv
- crafting valuable classes with operator overloading 604
- Craigslist** 43
- Craps Game Modification exercise 256, 301
- craps simulation 209, 210, 256
- crbegin**
 member function of containers 709
 member function of vector **719**
- crbegin** container member function **719**
- Create a New Project dialog in Visual Studio Community Edition 29
- create a sequential file 379
- create an array object from a built-in array or an initializer list 324
- create an object (instance) 85, 433
- create your own data types 75
- CreateAndDestroy** class
 definition 465
 member-function definitions 465
- creating xxviii
- creating algorithms xl
- Creating Three-Letter Strings from a Five-Letter Word exercise 302
- credit limit on a charge account 136
- Credit Limits exercise 136
- crend**
 member function of containers 709
 member function of vector **719**
- crend** container member function **719**
- crime prevention 53

- CRISPR (Clustered Regularly Interspaced Short Palindromic Repeats) gene editing 53
- critical section **1016**, 1019, 1020, 1027, 1059
- critical sections 1020
- crop yield improvement 53
- crossword-puzzle generator 415
- cryptocurrency 46, 51, 64, 129, 507, 1041
- cryptogram 410
- exercise 410
- cryptography xxix, xxxii, xlvi, 46, 494
- ciphertext 497
- Elliptic Curve Cryptography (ECC) **528**
- plaintext 497
- cryptographypublic-key xliv
- cryptographysecret-key xliv
- `<cstdint>` header 128, 330
- `<cstdio>` header 202
- `<cstdlib>` header 201, 679
- `<cstring>` header 201
- CSV (comma-separated values)
- .csv file extension **390**
- file format 358, **390**
- rapidcsv header-only library 390
- .csv filename extension 421
- CTAD (class template argument deduction) **265**, 719
- `<ctime>` header 201
- `<Ctrl>-d` 162, 380, 1114
- `<Ctrl>` key 162
- `<Ctrl>-z` 162, 380, 1114
- Cubing the Elements of a span exercise 347
- curly braces in format string 153
- current position in a stream 383
- current technology trends xxviii
- cursor **69**
- custom character class **402**
- custom concept 868
- Custom Concept exercise 930
- custom deleter for a `shared_ptr` 1147
- custom deleter function **1147**
- custom exception class 658
- custom functions xxix
- custom stream manipulator 1115
- customer
- service agents 53
- customization point 1162
- for derived classes 1162
- .cxx extension 25
- cybersecurity xl
- D**
- d presentation type **1130**
- Dall-E 2 xlvi, 16, 58
- dangling-else problem 139, **139**, 139, 140
- dangling pointer **620**
- dangling pointers 1148
- dangling reference **221**
- Dangling-else Problem exercise 139, 140
- dApps (decentralized applications) 47
- data 4
- categorical **414**
- getting to know xxxi
- mutable **989**
- numerical 414, 415
- data analytics xxxi, 358, 390, 392
- data counter (Simple compiler) **765**
- data hierarchy xxviii, **10**
- data-interchange format JSON 515
- data member **24**
- data mining **12**
- data munging **412**, **427**
- data persistence **358**
- data race **1015**
- data samples 418
- data science xxvii, xxviii, xl, 53, 359, 390, 412, 418, 429, 751
- get to know your data 392, 394, 429
- use cases 53
- data science curriculum proposal xl
- Data Science Project exercise
- Working with the `iamonds.csv` Dataset 414
- Working with the `iris.csv` Dataset 415
- data scientist 427
- data structure xl, **260**, 698
- data types
- char 199
- float 199
- int **72**
- long double 199
- long int 199
- long long 199
- long long int 199
- unsigned char 199
- unsigned int 199
- unsigned long 199
- unsigned long int 199
- unsigned long long 199
- unsigned long long int 199
- unsigned short 199
- unsigned short int 199
- data visualization 53
- data wrangling **412**, 427
- `data.gov` datasets 414
- database xl, **11**, 1036
- dataset 358, 390, 414, 418
- awesome-public-datasets 414
- `data.gov` 414
- diamonds **414**
- Iris 415
- Kaggle competition site 414
- Rdatasets 414
- repositories 414
- Titanic* disaster 392

- date and time utilities 993
- Date class 475, 512
 - exercise 509
- dates 191
- Davis, Jeffrey xlviii
- DbC (design by contract) **687**
- De Morgan's Laws exercise 186
- deadlock **1001**, 1069
 - four necessary conditions 1002
 - prevention (Havender) 1002
 - process or thread 1001
 - sufficient conditions 1002
- Deadlock and Indefinite Postponement discussion exercise 1069
- deallocate memory **598**, 600
- Debug area (Xcode) 38
- dec stream manipulator 1111, **1112**, **1119**
- decay to a pointer (array names) **322**
- decentralized applications (dApps) **46**, 47
- decentralized finance (DeFi) 46
- decimal digit **10**
- decimal (base 10) format 1130
- decimal (base-10) number system 1111
- decimal numbers 1119
- decimal point 117, 1105
- decision 101
 - making xxviii
 - symbol in the UML **101**
- declaration **72**
- declarative programming **284**
- decl type operator 1186
 - expression 1187
- decrement
 - a pointer 331
 - operator, -- **125**, 631
- decrypt 142
- decrypter 410
- deduction guide 894
- deep **620**
- deep copy **620**
- deep fakes 63
- deep learning **56**, 63, 359, 390
- DeepBlue **55**
- deeply nested statement 178
- default arguments **222**, 457
 - with constructors 457
- default assignment operator **470**
- default case in a switch **162**, 164, 207
- default constructor **441**, 449, 457, 479, 708
- default contract level **688**
- default copy constructor 478
- default delimiter 1109
- default destructor 463
- default special member function **557**, 557, 619
 - autogenerate a virtual destructor **557**, 619
- default type argument 821
 - for a type parameter **898**
- default type arguments for function template type parameters 898
- default violation handler (contracts) **690**
- default_random_engine **204**
- defect report (DR 1189)
- DeFi (decentralized finance) 46
- #define preprocessing directive **937**
- definite repetition **107**
- definition **147**
- Deitel & Associates, Inc. 1
 - virtual and on-site corporate training 1
- Deitel, Dr. Harvey M. 1
- Deitel, Paul J. 1
 - live instructor-led training xlvii
- delegating
 - constructor **463**
 - to other functions **852**
- delegating constructor **463**
- delete **598**, 602
 - placement 598
- delete[] (dynamic array deallocation) 599
- deleted constructor 1169
- deleter function **1147**
 - customize for a shared_ptr 1147
- deleting dynamically allocated memory 599
- delimiter (with default value '\n') 1107
- dependency injection **576**
- dependent condition 170
- dependent variable **420**, 420
- deprecated 200
- deque class template 704, 715, **728**, 851, 898
 - push_front function 728
 - shrink_to_fit member function **719**
- <deque> header 201, **728**
- dereference
 - a pointer **314**, 317, 326
 - an iterator 710, 711, 714
 - an iterator positioned outside its container 718
- dereferencing operator (*) **314**
- derive one class from another 474
- derived class **530**, 534
 - catch 682
 - customization point 1162
 - pointer to a base-class object 547
 - pointer to a derived-class object 547
- descriptive xxxi
- descriptive statistics xxxi, **395**, 395, 414, **418**, 418, 751
- deserializing data **516**

- design by contract (DbC; Bertrand Meyer) **687**
- design pattern **48**, 600
- design process **25**
- designated initializer (aggregates) **493**
- destructor **xxxiii**, 441, **463**, 589, 605, 708
 - called in reverse order **544**
 - called in reverse order of constructors **464**
 - in a class hierarchy **544**
 - should not throw exceptions **672**, 675
- destructor in a derived class **544**
- detach a thread **1007**
- developing algorithms **xxix**
- development environments **xl**
- device driver
 - polymorphism in operating systems **559**
- devirtualization **558**
- diagnostics that aid program debugging **202**
- diamond in the UML **97**, 186
- diamond inheritance (in multiple inheritance) **1174**
- Diamond-Printing Program
 - exercise **186**
 - modified **187**
- diamonds dataset **414**
- dice game **209**
- die rolling
 - exercise **300**
 - using an array instead of switch **274**
- difference_type** **707**
 - nested type in an iterator **887**
- digit **72**
- digit separator ' **207**
- digital ownership **46**
- direct access elements of a container **704**
- direct base class **533**, 533
- directly reference a value **312**
- disable assertions **671**
- Discord server **#include <C++>** lvii
- disjunction in a C++20 constraint or concept **862**
- disk **27**
- disk drive **1102**
- disk space **677**, 679
- dispatch a thread **1000**
- display a line of text **66**
- display screen **1102**
- Displaying Strings That End with ed exercise **409**
- Displaying Strings That Start with b exercise **409**
- distance** algorithm **872**
 - std::ranges** **883**
- Distance Between Points exercise **255**
- distribution (random-number generation) **204**
- distribution pass in bucket sort **1222**
- divide and conquer **230**
- divide by zero **27**, **113**, **354**
- DivideByZeroException** **663**
- divides** function object **822**
- division **6**, **76**, **77**
 - by zero **658**
 - compound assignment operator, /= **125**
- do...while** iteration statement **98**, **158**, 179
- Docker **xxxvii**, **lv**, **4**
 - clang++** container **xxxvii**, **28**, 935
 - container **xxxvii**, **lv**, **935**
 - containers **xxxvii**, **4**
 - Docker Desktop **35**, **36**
 - Docker Desktop installer **lv**
 - Docker Engine **35**, **36**
 - Docker Hub account **lv**
 - GCC Docker container **35**
 - GNU g++ container **xxxvii**, **28**, **34**, **35**
 - image **lv**
- Docker container **liv**
- documentation
 - C++ **xlvi**
- DOS (Disk Operating System) **15**
- dot operator (.) **85**, 456, 483, 553, 602
- dotted line in the UML **98**
- doub1e** data type **72**, **114**, 198
- double-ended queue **728**
- double-precision floating-point number **156**
- double quote **69**
- double-selection statement **98**, 120, 179
- "doubly initializing" member objects **479**
- doubly linked list **704**, **704**, 723
- download code **liii**
- download code examples **xxiii**
- driver program **83**
- drop range adaptor (C++20) **829**, **832**
- drop_while** range adaptor (C++20) **829**, **832**
- dual-core processor **8**
- duck typing **xxxvi**, **1142**, **1154**
- dump **352**
- duplicate elimination
 - exercise **299**, **843**
- duplicate keys **730**
- Duplicate Word Removal exercise **752**
- duration_cast** function template **993**
- dynamic
 - driving routes **53**
- dynamic binding **553**, 569, 570, 573
- Dynamic Binding vs. Static Binding exercise **582**
- dynamic data structure **310**

- dynamic memory allocation
344, 589, **598**, 599, 600,
601, 669, 677
array of integers 615
- dynamic storage duration
1182
- dynamic_cast** 681
- dynamically determine function to execute 552, 554
- E**
- E** (or **e**) presentation type
1129
- Easylogging++ logging library
685
- Eclipse
Foundation **15**
IDE 26
- edge computing **43**
- edit a program 25
- editor 25
- Editor** area (Xcode) 38
- EEPs (examples, exercises and projects) xlivi
- efficiency of
binary search 701, 1207
bubble sort 1222
linear search 701, 1202
merge sort 1219
selection sort 1213
- Eight Queens
Brute force approaches exercise 307
exercise 306
with recursion exercise 307
- electronic health records 53
- element of an array **261**
- elements** range adaptor (C++20) 829, 834
- Elliptic Curve Cryptography (ECC) **528**
- Elliptic Curve Cryptography project exercise 528
- else** keyword 102
- emacs** editor 25
- e-mail (electronic mail) 41
- embedded parentheses **77**
- embedded system xxiii, xliv, 7, 16, 18, 670
- emotion detection 53
- emplace** member function
of containers 709
of queue 742
of stack 740
- emplace_after** 709
- emplace_front** 709
- emplace_hint** 709
- Employee** abstract base class 560
- Employee** class 475
definition showing composition 476
definition with a **static** data member to track the number of **Employee** objects in memory 488
exercise 509
header 562
implementation file 563
member-function definitions 477, 489
- employee identification number 11
- empty** member function
of containers 709
of **priority_queue** 743
of queue 742
of sequence container **722**
of stack 740
of string **86**, 365, **592**
- Empty Project** template 29
- empty statement 104, 147, 149
- empty string 85, 365, 434
- enable compiler warnings 174
- encapsulation **24**, 437, 470
- enclosing scope **667**
- encryption 142, 410
- end**
function **278**, 323
function of header <array> **278**
- member function of containers 708, **710**
- end of a stream 383
- “end of data entry” 112
- end-of-file (EOF)
indicator **162**, 380, 381, 1125
- key combination 381
- marker **377**
- ends_with** member function of class **string** (C++20) **86**
- Energy Sciences Network terabit Internet 41
- Enforcing Privacy with Cryptography exercise 142
- engine (random-number generation) **204**
- English-like abbreviations 13
- Enhancing Class Date exercise 512
- Enhancing Class Time exercise 512, 515
- ensures** contract keyword **688**, 689
- Enter* key 74
- entry point 176
- enum class** **210**
- enumeration **210**
constant **211**
- EOF 1106, 1109
- eof** member function 1106, 1125
- eofbit** of stream **1125**
- equal** algorithm 615, **627**, 837
ranges version (C++20) **783**, **784**
- equal** to 79
- equal_range**
algorithm 837
algorithm, ranges version (C++20) 810
function of associative container **733**
- equal_to** function object 822
- equality operators **79**, 80
!= 100
== 604
== and != 100

- EqualityComparable 710
 equation of straight line 77
erase 370
 algorithm from header `<vector>` 788
 member function of containers 709
 member function of first-class containers 722
 member function of `string` 370
 member function of `vector` 787
 erase-remove idiom 787, 787, 788
erase_if algorithm from header `<vector>` 788
 e-reader device 17
 error detected in a constructor 672
 error state of a stream 1124
error_code class 684
 escape character 69
 escape sequence 69, 70, 389
 `\' (single-quote character) 69
 `\" (double-quote character) 69
 `\\ (backslash character) 69
 `\a (alert) 69
 `\n (newline) 69
 `\r (carriage return) 69
 `\t (tab) 69
 eText (Pearson) xlvii
 Ether (cryptocurrency of Ethereum platform) 47
 Ethereum 47, 63
 ethics xl, xlv, 63
 Euler's totient function 523
 Evaluate Word Problems exercise 413
 even integer 184
 Even Numbers exercise 251
 exabytes (EB) 50
 exaflops 7, 9
 examples (download) xxiii
 examples, exercises and projects (EEPs) xlivi
 exception 264, 294, 295, 655
 bad_alloc 598
 handler 294
 handling 290
 in the context of constructors and destructors xxxiii
 invalid_argument 373
 memory footprint 656
 out_of_bounds 604
 out_of_range 295, 373
 parameter 295
 safety xxxiii
 what member function of an exception object 295
 when to use xxxiii
exception class 681, 681
 what virtual function 661
exception guarantee
 copy-and-swap idiom 664
exception handling 201, 655
 flow of control 656, 694
<exception> header 201, 681
 exception in a thread 1003
 exception parameter 661
 exception-safe code 664, 1143
 exception safety guarantees
 basic exception safety guarantee 664
 no guarantee 664
 no throw exception safety guarantee 664
 strong exception safety guarantee 664
 exception types in the C++ standard libarary 681
 exceptions
 bad_alloc 676
 bad_cast 681
 bad_typeid 681
 filesystem_error 684
 length_error 682
 exceptions (cont.)
 logic_error 681
 out_of_range 682
 overflow_error 681
 underflow_error 681
exchange function (header `<utility>`) 623
 exclusive 1008
 exclusive resource 1008
exclusive_scan
 algorithm 838
 parallel algorithm 998
 executable
 image 27
 program 26
 execute a program 25, 27
 execution
 parallel 996
<execution> header 202, 994
 execution policy 994, 995
 std::execution::par 994
 execution-time error 28
 execution-time overhead 570
 executor (`concurrency`)
 coroutine support library 1081, 1084
 scheduling tasks 1076
 exit a function 69
exit function 380, 464, 679
 exit point 176
 of a control statement 99
EXIT_FAILURE 380, 679
EXIT_SUCCESS 380
 exited block 1182
 exiting a `for` statement 167
exp function 192
expects contract keyword 688, 689, 694
 expiring value 611
explicit
 narrowing conversion 199
explicit keyword 440, 643
 constructor 642
 conversion operators 643
 exponent 651

exponential function 192
 exponential notation 1129,
1129
 exponentiation 156
 exercise 250
 modular 526
export (C++20 modules)
 a block 942
 a declaration **942**, 942,
 945
 a namespace 942
 a namespace member 942
export import (C++20
 modules) 960
export module (C++20
 modules) **944**
 declaration for a module
 interface partition 960
 expression 100, 116
 extended reality **46**
 extensible **531**, 545
 programming language
432
extern keyword 1183
extern storage-class specifier
1182
 external iteration **268**
extract member function of
 associative containers 709
 extracting data from text 400

F

f presentation type 1129
fabs function 192
 Facebook 16
 facial recognition 53
 factorial 141, 184, 230
 with **partial_sum** **816**
 Factorial exercise 141
factorial function 230
 factorials 816
 fail fast 671
fail member function **1125**
failbit of stream 1110,
1124, 1125
 fair rounding algorithm 180

[[fallthrough]] attribute
163
false 80, **100**, 1122
 fatal error **104**, 354
 fatal logic error **104**
 fatal runtime error **28**
 fault-tolerant programs **294**,
 655
 features in a dataset **392**
 Fertig, Andreas xlix
 fetch 352
 fetch the next instruction **353**
fibonacci function 236
 Fibonacci series 234, 236,
 1076
 defined recursively 234
 generator coroutine 1077
 Fibonacci Series exercise 253
field **11**, 11
 field alignment 1130
 field width **155**, 1111, 1114,
 1130
 fields larger than values being
 printed 1118
 FIFO (first-in, first-out) 706,
 728, 742
file **11**, **358**, 382
file open mode **379**, 381
 ios::app **380**
 ios::ate 380
 ios::binary 380
 ios::in 380, **382**
 ios::out **379**
 ios::trunc 380
 file-position pointer **383**
 file processing 191
 file scope **240**, 455, 945
 filename **379**, 381
 filename extensions 25
 .cpp 944
 .cppm 944
 .h 434
 .i 944
 .ixx **944**, 944
 .pcm 944
<filesystem> header 202,
380, 684

filesystem_error class
 684
filesystem::path **380**
fill algorithm 837
 ranges version (C++20)
781, 782
 fill character 1111, **1114**,
 1118, 1130
fill member function 1117,
1118, 1126
 fill with 0s 1131
fill_n algorithm 837
 ranges version (C++20)
781, 782
 filter 828
filter range adaptor
 (C++20) 829
 filtering in functional-style
 programming **287**, 828
final
 class 557
 member function **557**,
 558
 final state in the UML **98**,
 176
 final value 147
final_suspend function of
 a coroutine promise object
1095
find algorithm 837
 ranges version (C++20)
796, **796**
find function of associative
 container **732**
find member function of
 class **string** **368**, 368
find member function of
 string_view **377**
 Find the Error exercise 255,
 256
 Find the Largest exercise 137
 Find the Minimum exercise
 251
 Find the Minimum Value in
 an **array** exercise 308
 Find the Smallest Value exer-
 cise 184

- Find the Two Largest Numbers exercise 138
- find_end** algorithm 837
- find_first_not_of** member function of class *string* 369
- find_first_of** algorithm 837 member function of class *string* 368
- find_if** algorithm 837 ranges version (C++20) 796, 797
- find_if_not** algorithm 837 ranges version (C++20) 796, 799
- find_last_of** member function of class *string* 368
- finding strings and characters in a *string* 367
- first** data member of *pair* 733 member function of *span* class template (C++20) 338
- first-class container
- **begin** member function 710
 - **clear** function 723
 - **end** member function 710
 - **erase** function 722
- first refinement in top-down, stepwise refinement 112, 119
- first-in, first-out (FIFO) 706, 728
- data struture 742
- fixed notation 1105, 1111, 1120
- fixed point
- **format** 117
- fixed-size data structure 320
- fixed-size integer types 330
- fixed stream manipulator** 117, 1111, 1113, 1116, 1120
- fixed-size integer types 330
- flag value 112
- flagged 761
- flags** member function of *ios_base* 1123
- float** data type 114, 199
- floating point 1111, 1120
- scientific format 1120
- floating-point arithmetic 588
- floating-point literal 154, 156
- **double** by default 156
- floating-point number 110, 114, 116
- **double** data type 114
 - double precision 156
 - **float** data type 114
 - single precision 156
- floating_point** concept 861, 868
- floating-point number formatting 155
- floor** function 192, 249, 250
- FLOPS (floating-point operations per second) 9
- flow of control 116
- exception handling 656, 694
 - **if...else** statement 102
 - virtual function call 572
- flush buffer 1127
- flushing stream 1111
- fmod** function 192
- **fmodule-file** compiler flag (clang++) 948
- **fmodules-ts** compiler flag (g++) 940
- fmtflags** data type 1123
- fold expression 848, 902, 905
- binary left fold 905, 906
 - binary right fold 906
 - exercise 930
 - unary left fold 906
 - unary right fold 906
- fold operation
- binary left 909
 - binary right 909
 - unary left 908
 - unary right 908
- folly::coro** coroutine support library 1076
- font conventions in this book xlii
- for** iteration statement 98, 148, 148, 156, 179
- header 149
- for_each** algorithm 615, 631, 837
- ranges version (C++20) 779
- for_each_n**
- algorithm 837
 - parallel algorithm 998
- force a decimal point 1105
- force a plus sign 1118
- format error 1125
- format** function from header <*format*> (C++20) 152, 155, 1127
- <*format*> header (C++20) 203
- **format** function 152, 155, 1127
- format of floating-point numbers in scientific format 1120
- format specifier (C++20 text formatting) 155
- format state 1111, 1123
- format string 153
- curly braces in a replacement field 153
- formatted I/O 385, 1103
- formatted text 385
- formatting 155
- type dependent 1129
- formatting (C++20)
- **d** (integer) 156
 - **f** (floating-point number) 155
 - field width 155
 - format string 153
 - placeholder 153, 155
 - precision of a floating-point number 155
 - right align > 155
 - strings 152

- formatting strings 1127
 formulating algorithms 107
 forums
 groups.google.com/g/comp.lang.c++ xlvi
 reddit.com/r/cpp/ xlvi
 stackoverflow.com xlvi
 forward 1148
 forward class declaration 1148
 forward iterator 712, 724,
 776, 801
 operations 714
 forward reference (Simple compiler) 761
forward_iterator concept (C++20) 777, 787, 794
forward_list class template 705, 715, 724
 splice_after member function 725
<forward_list> header 201, 724
forward_range concept (C++20) 777, 787, 789, 794, 798, 804, 810, 811
 four V's of big data 52
 fprebuilt-module-path-
 fmodule-file compiler flag (clang++) 948
Fraction class exercise 651
 fragile
 base-class problem 1161
 software 1161
 fraud detection 53
 free function 451, 481, 635
 free store 598, 599
friend
 of a base class 1160
 of a derived class 1160
friend function 480
 can access **private** members of class 480
 friendship granted, not taken 480
friendship
 not symmetric 480
 not transitive 480
front
 member function of queue 742
 member function of sequence containers 721
 member function of span class template (C++20) 338
 member function of vector 396
front_inserter function template 789
fstream 378
<fstream> header 201, 378
 full template specialization 917
 function xxix, 19, 23, 27, 68
 anonymous 286
 call overhead 217
 call stack xxix, 214, 233
 call/return mechanism xxix
 constexpr 919
 constexpr 919
 definition 240
 free 451, 635
 header 195
 hypot 192
 name 1183
 overloading 224
 parameter 195
 parameter list 195
 prototype 195, 197, 219,
 240
 signature 198, 225
 function call operator () 644, 1192
 function object 730, 730, 736, 746, 775, 821
 also called a functor 775, 821
 arithmetic 822
 binary 823
 divides 822
 equal_to 822
 greater 822
 greater_equal 822
 function object (cont.)
 less 822
 less_equal 822
 less<int> 730
 less<T> 736, 743
 logical 822
 logical_end 822
 logical_not 822
 logical_or 822
 minus 822
 modulus 822
 multiplies 822
 negate 822
 not_equal_to 822
 plus 822
 predefined in the STL 822
 relational 822
 function overloading 292
 function parameter scope 239
 function pointer 344, 571, 775, 823
 function prototype 195, 480
 in a class definition 449
 function scope 239
 function template 227, 847, 872
 abbreviated (C++20) 854, 854
 maximum 256
 maximum exercise 256
 minimum 256
 minimum exercise 256
 unconstrained 857
 Function Template exercise 929
 function template specialization 227, 228, 229
 function try block 673, 674, 675
 Functional 844
<functional> header 202, 822
 functional programming 918
 functional structure of a program 68

- functional-style programming xxvii, 18, 203, 287, 615, 775, 828
array of Invoices exercise 844
 Calculating Employee Average Salaries by Department exercise 844
 Cubing Integers exercise 303
 filtering 287
 Filtering and Sorting exercise 844
 mapping 288
 reduction 271, 284
 Summing the Triples of the Even Integers from 2 through 10 exercise 303
 functor (function object) 775, 821
 fundamental type xxviii, 72, 128
 bool 100
 char 72, 199
 double 114
 float 114
 int 125
 long 128, 128
 long double 114, 156
 long long 128, 129, 232
 promotion 117
future class template
 get member function 1047
 <**future**> header 202, 1045
- G**
- G** (or **g**) presentation type 1129
g++ compiler xxviii, 32
 in a Docker container 28
game of “guess the number” 252
game of chance 209
game of craps 210
game playing 203
 game systems xxiii, xliv, 18
 garbage value 441
 Gary Kasparov 55
 Gas Mileage exercise 135
 gathering pass in bucket sort 1222
 GCC Docker container 35
gcd algorithm 254, 813, 813, 838
gcount function of **istream** 1110
GDPR (General Data Protection Regulation) xlv, 63
 general class average problem 111
 General Data Protection Regulation (GDPR) xlv
 generalities 545
 generalized numeric operations 836
generate algorithm 837
 ranges version (C++20) 781, 782
generate_n algorithm 837
 ranges version (C++20) 781, 783
 Generating Integer Ranges with Coroutines exercise 1099
 Generating Tuples with Coroutines exercise 1099
 generating values to be placed into elements of an array 269
 generative AI 57
 Generative Pre-Trained Transformer (GPT) large language model 57
generator coroutine 1077, 1079
 Fibonacci sequence 1077
generator coroutine support library 1076
 Sy Brand 1077
 t1::generator class template 1077
generator function 781
 generic algorithms 776
 generic lambda 286, 286, 324, 780, 1159
 generic lambdas 286
 generic programming xxvii, 18, 227
get function for obtaining a tuple member 901
get member function 1106, 1107
get member function of a unique_ptr 618
get member function of class template future 1047
get pointer 383
 get to know your data 392, 394, 419, 429
get_id function of the **std::this_thread** namespace 1004
get_return_object function of a coroutine promise object 1094
getline function of **cin** 1109
getline function of the string header 247
 gets the value of 79
 getting questions answered xlvi
 getting to know your data xxxi
 gigabytes (GB) 6, 50
 gigaflops 9
 Git liii
 Git version control system 48
 GitHub xxvi, xxxviii, xxxix, xliv, xvii, liii, 15, 20, 48
 C++20 Standard Document xlvi
 global 451
 global function 192
 global module 951
 fragment 951
 global namespace 1180
 global namespace scope 239, 240, 464, 945, 946

global object constructors 464
 global scope 464, 466, 1180
 global variable 240, 240, 241, 243, 244, 1180, 1183
 GNU C++ xxxvii, liv, 28 g++ xxviii, 3
 GNU Compiler Collection (GCC) Docker container 28, 34, 35, 935
 Standard Library Reference Manual xlvi
gnuplot xxxi, 418
 install 421
 Go board game 56
 Go programming language 22
godbolt.org
 Compiler Explorer website 689
 golden mean (Fibonacci) 234
 golden ratio (Fibonacci) 234
 good function of `ios_base` 1126
 Google Assistant xlv, 63
 Google Bard 58
 Google C++ Style Guide 683
 Google Cloud Platform (GCP) 42
 Google Concurrency Library (GCL) concurrent containers 1062
 Google Logging Library (glog) 685
 Google Maps 43
 Google Search xl ix
 Google Translate 413
 Gosling, James 21
`goto` elimination 96, 97
`goto` statement 96
 GPS (Global Positioning System)
 device 5

GPT (Generative Pre-Trained Transformer) large language model 57
 GPU (graphics processing unit) 991
 Grammarly xl ix
 graph information 185, 272
 graphical user interface (GUI) 16
 graphics processing unit (GPU) 8
`greater` function object 822
`greater_equal` function object 822
 greater-than operator 79
 greater-than-or-equal-to operator 79
 greatest common divisor (GCD) 252, 254, 526, 813
 exercise 252
 greedy evaluation 828, 1076
 greedy quantifier 403
 Gregoire, Marc xl ix
 gross pay 137
 grouping (operators) 77
 grouping not changed by overloading 597
groups.google.com/g/comp.lang.c++ xl vi
`<gs1/gs1>` header 200
 guard condition in the UML 101
 guarding code with a lock 1020
 Guess the Number Game exercise 252
 Guess the Number Game Modification exercise 252
 GUI (Grahical User Interface) 16
 Guidelines Support Library (GSL) 199, 321
 Guido van Rossum 21

H

.h filename extension (header) 434
 Hadavi, M. Michael xl viii
 half-open range 287, 714
 half-up rounding 182
 hallucinations in large language models 58
 halt 352
 halt instruction 761
 handle on an object 456
`handle_contract_violation` default contract violation handler 690
 hardcopy printer 27
 hardware xxiv, xxviii, 2, 4, 13
 hardware platform 18
has-a relationship 474, 530
 hash (hashable keys) 730
 hash bucket 703
 hash table 703
 hash-table collisions 703, 704
 hashable 734, 738
 type requirements 730
 hashing 703, 730
 Havender (deadlock prevention) 1002
 head of a queue 699
 header 200, 449, 685
 .h file 434, 434
 header of a function 195
 header-only library 200, 938
`inline` variable 899
 header unit (C++20 modules) 938, 939, 940, 947
 compile a header 940
 precompile a header 940
 headers 374
`<algorithm>` 618, 627, 692, 720, 774, 837
`<array>` 262
`<atomic>` 1048
`<barrier>` (C++20) 1055
`<cassert>` 671
`<chrono>` 448, 993
`<cmath>` 156, 192, 193

- headers (cont.)
 <compare> 637
 <concepts> **861**
 <condition_variable> **1019**
 <coroutine> (C++20) **1094**
 <cstdint> 128, 330
 <cstdlib> 679
 <deque> **728**
 <exception> **681**
 <execution> **994**
 <filesystem> **380**, 684
 <forward_list> 724
 <functional> **822**
 <future> **1045**
 <gsl/gsl> 200
 <initializer_list> **618**
 <iomanip> 117
 <iostream> **67**
 <latch> (C++20) **1052**
 <limits> 110, 132
 <list> **723**
 <map> **736**, 738
 <memory> **601**, 774, **838**, **1143**
 <mutex> **1019**, 1036
 <numbers> **193**
 <numeric> 774, 813, **838**
 <queue> **742**, 743
 <random> **203**
 <ranges> **286**
 <regex> **400**, 405
 <semaphore> (C++20) **1059**
 <set> 730
 <stack> **740**
 <stdexcept> **658**, 681
 <stop_token> (C++20) **1037**
 <string> **85**
 <thread> **1003**
 <tuple> **899**
 <type_traits> 864, 921
 <unordered_map> 736, 738
 headers (cont.)
 <unordered_set> 731, 734
 <utility> **623**
 <variant> 1154
 <vector> 290
 Health Insurance Portability and Accountability Act (HIPAA) xlv
 health outcome improvement 53
 heap 743, 816, 817
 max heap 816
 min heap 816
 heapsort
 make_heap algorithm **818**
 pop_heap algorithm **819**
 push_heap algorithm **819**
 sort_heap algorithm **818**
 sorting algorithm **817**
 helper function **457**
 heterogeneous lookup (associative containers) **734**, 1193
 heuristic **305**
 heuristic programming xxx
 hex stream manipulator **1111**, **1112**, **1119**
 hexadecimal
 integer 315
 hexadecimal (base-16) number 1105, 1111, 1112, 1119, 1130
 hide a data member 482
 hide implementation details 437
 hide names in outer scopes 240
 hierarchical relationship 533, 534
 hierarchy
 of employee types 535
 of exception classes 680
 high-level concurrency features 1076
 high-level language **13**
 higher-order functions **284**
 highest level of precedence 77
 “highest” type 199
 high-level I/O 1103
 HIPAA (Health Insurance Portability and Accountability Act) xlv, 63
 hold-and-wait condition 1002
 Hollman, Dr. Daisy xliv
 hook (Simple compiler) **761**
 horizontal tab ('`t') 69
 HTML (HyperText Markup Language) **42**
 HTTP (HyperText Transfer Protocol) **42**
 HTTPS protocol 129, 246
 Huge integers 650
 HugeInt class 647
 human genome sequencing 53
 HyperText Markup Language (HTML) **42**
 HyperText Transfer Protocol (HTTP) **42**
 hypot function **192**
 hypotenuse
 in three-dimensional space 192
 of a right triangle 192, 193
 hypotenuse 196, 250
 Hypotenuse Calculations exercise 250

I

- I/O completion 669
 IBM DeepBlue **55**
 IBM Watson **55**
 IDE (integrated development environment) **26**
 identifier **73**, 98, 240
 linkage 1183
 identifiers for variable names 1182
 identity theft prevention 53
 IEEE 754 floating-point standard 691

if single-selection statement **79**, 98, 100, 179
 with initializer 165
if...else double-selection statement **98**, **101**, 102, 116, 179
 with initializer 165
.ifc filename extension 944
ifstream 378, 381, 382
ignore function of **istream** 1109
IGNORECASE regular expression flag **406**
image (Docker) **lv**
imitation game **58**
immutable **283**
 data **1016**
 data and thread safety 989
 keys 705
 string literal **341**
 implement an interface 575
 implementation inheritance 543, 1154
 implementation of a member function changes 462
 implementation of merge sort 1214
 implicit conversion **117**, 440, 641, 642
 improper 641
 user defined 641
 via conversion constructors 642
 implicit first argument 482
 implicit handle 456
import statement (C++20 modules) **946**
 existing header as a header unit **938**
 improve performance of bubble sort 1222
 in-class initializer **449**
 in-memory
 formatting 386
 I/O **386**
 in parallel **988**
 include guard **450**, 946
#include <iostream> 67
includes algorithm 837
 ranges version (C++20) 807, **808**
 including a header multiple times 449
inclusive_scan
 algorithm 838
 parallel algorithm **998**
 increment 149
 a control variable **147**, 147
 a pointer 331
 an iterator 714, 884
 expression 168
 operator 631
 indefinite postponement 1000, **1001**, 1002, 1069
 indentation 102, 103
 independent variable **420**, 420
index **261**
 indexed access 728
 indexed name used as an *rval_ue* 610
 indirect base class **533**, 533
 indirect recursion **232**
 indirection **312**
 operator (*) **314**
 triple 570
 indirectly reference a value **312**
indirectly_copyable concept (C++20) 779
indirectly_readable concept (C++20) 779
indirectly_swappable concept (C++20) 801
indirectly_writable concept (C++20) 779, 790
 inequality 1185
 operator keywords 1185
 inequality operator (!=) 604
-inf (negative infinity) 658
inf (positive infinity) 658
 infer (determine) a variable's data type **281**
 infer a lambda parameter's type 286, 324, 780
 infinite loop **106**, 141, 149, 150, 151, 233
 infinite range **830**
 infinite recursion **233**, 238
 infinite sequence 1076
 infinite series for calculating π 186
 infix notation **754**
 infix-to-postfix conversion **754**
 Infix-to-Postfix Converter exercise 754
 information hiding **24**, **437**
 inheritance **24**, 474, 530, **530**, 530, 534
 constructors 1168, 1169
 hierarchy **533**
 implementation 582, 1154
 interface **574**, 582, 1154
 members of an existing class **530**
 multiple **1169**, 1170, 1171
public **534**
virtual **1176**
 Inheritance Advantage exercise 582
 inheriting interface versus implementation 582
 Inheriting Interface vs. Implementation exercise 582
 initial state in the UML **98**, 176
 initial value of control variable **147**
initial_suspend function of a coroutine promise object **1094**
 initialization
 once-time, thread-safe 1047
std::call_once **1048**
std::once_flag **1048**
 initialize a pointer 312

- initializer **265**
- initializer list **265**
- initializer_list* class template **618**, 618, 707, 732
 - size member function 618
- <*initializer_list*> header **618**
- initializing
 - an array's elements to zeros and printing the array **262**
 - multidimensional arrays **280**
 - the elements of an array with a declaration **265**
- inline** **218**
 - function **217**, 218
 - keyword **217**
 - variable **488**, 899
- inline_executor** (concurrent) **1085**, 1085
- inner block **240**
- inner_product** algorithm **838**
- innermost pair of parentheses **77**
- inplace_merge** algorithm **837**
- ranges version (C++20) **805**, **806**
- input xxviii
- input a line of text **1109**
- input and output stream iterators **711**
- input device **5**
- input from string in memory **202**
- input iterator **712**, 713, 714, **778**
- input/output (I/O) **191**
 - header <*iostream*> **67**
 - library functions **202**
- input sequence **710**
- input stream **1106**, 1107
- input stream iterator **711**
- input stream object (*cin*) **74**
- input unit **5**
- input_iterator** concept (C++20) **777**, 788, 805, 873
- input_or_output_iterator** concept (C++20) **783**
- input_range** concept (C++20) **777**, 779, 784, 785, 786, 788, 790, 791, 792, 793, 795, 796, 797, 798, 799, 801, 803, 804, 805, 806, 808, 809, 813
- input/output operators **349**
- Inputting Decimal, Octal and Hexadecimal Values **1138**
- inputting from *strings* in memory **386**
- insert**
 - at back of vector **715**
- insert**
 - function of associative container **733**
 - function of containers **709**
 - function of *multimap* **737**
 - function of *multiset* **731**
 - function of sequence container **721**
 - function of *set* **735**
 - function of *string* **372**
- inserter** function template **789**
- insertion sort **1210**, 1213, **1219**
 - algorithm **1209**
 - efficiency **1208**
- instance **23**
- instance variable **436**, 445
- instantiate
 - class template **847**
 - template **847**
- instruction **27**, **756**
 - counter **765**
 - execution cycle **352**, 353
- Instructor Resource Center (Pearson) xlvi
- instructor resources for C++ *How to Program*, 9/e xlvi
- Instructor-Led Training with Paul Deitel xlvii
- int** & **219**
- int data type **68**, 73, 125, **198**
 - operands promoted to *double* **117**
- integer **68**, **72**, 140
 - arithmetic **588**
 - cpp_int* class from the Boost Multiprecision library **130**
 - division **76**, 110
 - super-sized **129**
 - types, fixed-size **330**
- integer formatting **156**
- integerPower** **250**
- integers prefixed with 0 (octal) **1119**
- integers prefixed with 0x or 0X (hexadecimal) **1119**
- IntegerSet** class **513**, 645
- integral concept **861**, 868
- integral constant expression **159**, 599
- integral expression **164**
- integrated development environment (IDE) **26**
- intelligent assistants xlvi, 53, 63
 - Amazon Alexa xlvi, 63
 - Apple Siri xlvi, 63
 - Google Assistant xlvi, 63
 - Microsoft Cortana xlvi, 63
- intelligent virtual assistants **425**
- inter-thread communication **1046**
 - std::future* **1046**
 - std::promise* **1047**
- intercept **420**
- interest on deposit **187**
- interest rate **153**
- interface **448**, 559, 574
 - dependency **984**
 - inheritance **561**, **574**, **1154**
 - of a class **448**
 - to a hierarchy **560**
- Interface Builder **16**

- interlanguage translation 413, **425**
- internal **284**
- internal iteration **269**
- internal linkage 945
- internal stream manipulator 1111, 1117
- International Standards Organization (ISO) 18
- Internet **41**
 - bandwidth 41
 - terabit 41
- Internet bandwidth xxiv
- Internet of Things (IoT) xxiv, 17, 42, 53, 63
- Internet Protocol (IP) **41**
- interpreter **14**
- Intro to Similarity Detection with Very Basic Natural Language Processing 424
- Intro to Similarity Detection with Very Basic Natural Language Processing case study exercise 424
- intToFloat** 196
- invalid operation code **354**
- invalid_argument** exception 373, **451**, 682
- invariant 686
 - class 686
- inventory control 53
- Invoice** class (exercise) 509
- invoke** function 827
- <iomanip> header 117, 201, 1103, 1112
- iOS 15, **16**
- ios_base** class 1124
 - precision** function **1113**
 - width** member function **1114**
- ios::app** file open mode **380**
- ios::ate** file open mode 380
- ios::beg** seek direction **383**
- ios::binary** file open mode 380
- ios::cur** seek direction **383**
- ios::end** seek direction **383**
- ios::in** file open mode 380, **382**
- ios::out** file open mode **379**
- ios::trunc** file open mode 380
- <iostream> header **67**, 201, 1103
- IoT (Internet of Things) 53
- iota**
 - algorithm 813, **814**, 838
 - range factory (C++20) 830
 - range factory (C++20), infinite range **830**
- IP address **41**, 42
- iPad **16**
- iPadOS **16**
- iPhone **16**
- Iris dataset 415
- is-a* relationship (inheritance) **530**, 1178
- is_arithmetic** type trait 875
- is_base_of** type trait 919
- is_heap** algorithm 838
- is_heap_until** algorithm 838
- is_partitioned** algorithm 838
- is_permutation** algorithm 837
- is_sorted** algorithm **692**, 838
- is_sorted_until** algorithm 838
- isEmpty** member function of a stack 852
- isnan** function of header <cmath> **395**
- Issue** navigator 38
- istream** class 383, 386
 - peek** function **1109**
 - seekg** function **383**
 - tellg** function **383**
- istream_iterator** **710**
- istringstream** class **386**, 387
- iter_swap** algorithm 800, **801**, 837
- iteration 99, 179, 237
 - of a loop 167
 - statement 237
- iteration statement xxix, **97**, 98, 105
 - do...while** 158, 179, 179
 - for** 179
 - while** **106**, 147, 179
- iteration terminates 106
- iterative (non-recursive) solution 230, 238
- iterator **698**
 - contiguous 712
 - minimum requirements 776
 - nested type names 887, 889
 - pointing to the first element of the container 710
 - pointing to the first element past the end of container 710
 - read/write 888
 - read-only 885
 - string** 786
 - type names 713
- iterator** 707, 708, 710, 713, 733, 735
- iterator adaptor **788**
 - back_inserter** **788**
 - std::reverse_iterator** **892**
- iterator concepts (C++20) 777
 - complete list 777
- <iterator> header 202, 788
- iterator operation 884
- iterator_category** nested type in an iterator 887
- .**ixx** filename extension **944**, 944

J

Jacopini, G. 96
 Java Platform Module System (JPMS) 973
 Java programming language 17, 21
 JavaScript programming language 22
 Jobs, Steve 16
join function of a `std::jthread` 1007
 joining a thread 1084
 Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards (2005) 684
 JSON (JavaScript Object Notation) 515, 516
 array 515, 516
 Boolean values 516
 cereal library 390
 data-interchange format 515
 false 516
 JSON object 515
 null 516
 number 516
 RapidJSON library 390
 serialization 516
 string 516
 true 516
`JSONInputArchive` (cereal library) 520
`JSONOutputArchive` (cereal library) 519
`jthread` class (C++20) 1003, 1003, 1008

K

Kaggle datasets 414
 Kalev, Danny Ph.D. xl ix
 kernel of an operating system 14
 key 730
 key-value pair 703, 705, 736, 737, 739
 keyboard 4, 27, 74, 378, 1102, 1104

keys range adaptor (C++20) 829, 833
keyword 68, 98
 and 1185
 and_eq 1185
 auto 281
 bitand 1185
 bitor 1185
 break 163
 case 162
 class 227, 435, 850
 co_await (C++20) 1074
 co_return (C++20) 1074
 co_yield (C++20) 1074
 compl 1185
 concept (C++20) 868
 const 207
 constexpr 270
 continue 167
 default 162
 do 98, 158
 else 98
 enum class 210
 explicit 440, 642
 extern 1183
 for 98, 148
 if 98
 inline 217
 mutable 1184
 namespace 945, 1180
 not 1185
 not_eq 1185
 operator 596
 or 1185
 or_eq 1185
 private 437, 437
 public 437
 static 213, 240, 1183
 switch 98
 template 850, 850
 thread_local 990
 throw 663
 typename 227, 850
 unsigned 198
 while 98, 158
 xor 1185
 xor_eq 1185

Knight's Tour xxx
 brute force approaches exercise 305
 closed tour test exercise 307
 exercise 303
 Kotlin programming language 17, 21
 Kühl, Dietmar xl ix

L

label in a `switch` 162
lambda 286, 775, 778
 capture variables 397, 633
 expression 286, 631
 generic 286, 324, 1159
 infer a parameter's type 286, 324, 780
 introducer 286, 397, 633, 780
 introducer [&] (capture by reference) 780
 introducer [=] (capture by value) 780
 templated (C++20) 856
 templatized 854
 language identification 425
 large language model 57
 GPT (Generative Pre-Trained Transformer) 57
 hallucination 58
 large language models 54
 last-in, first-out (LIFO)
 data structure 706, 740
 order 849, 852
 last member function of `span` class template (C++20) 338
 last-in, first-out (LIFO) 214
latch (C++20) 1052, 1052, 1053
`<latch>` header 203
 Latches exercise 1070
 late binding 553
 launch a long-running task asynchronously 1074

- `launch` enum **1046**
- `launch` enum **1046**
 - async **1046**
- launch policy (multithreading) **1046**
- Launching Tasks with `std::async` exercise **1070**
- lazily computed sequence (generator) **1075**
- lazy evaluation **287**, 828, **1076**
- lazy pipeline **288**
- `lcm` algorithm **813**, **814**, 838
- leading 0 **1119**
- leading 0x and leading 0X **1111**, 1119
- leaf node in a class hierarchy **560**
- least common multiple **527**, **814**
- ledger (blockchain) **46**
- left align **1117**
- left align (<) in string formatting **155**, 1130, 1131
- left brace ({}) **68**
- left fold
 - binary **909**
 - unary **908**
- left justified **102**
- left-shift operator (<<) **588**
- left side of an assignment **173**, **261**, 468, 610
- `left` stream manipulator **1111**, **1117**, 1117
- left-to-right evaluation **77**
- left value **173**
- left-align text **1131**
- legacy code **19**, 598
- lemmatization **426**
- `length` member function of class `string` **85**
- length of a string **341**
- `length_error` exception **365**, **682**
- `less` function object **822**
- less-than operator **79**
- less-than-or-equal-to operator **79**
- `less_equal` function object **822**
- `less<int>` **730**, 736
- letter **10**
- level of indentation **102**
- Levi, Inbal **xlix**
- lexicographical **362**, 363
 - comparison **86**, 591
 - sort **278**
- `lexicographical_compare` algorithm **837**
- ranges version (C++20) **783**, **786**
- `lexicographical_compare_three_way` algorithm **838**
- lifetime of an object **436**
- LIFO (last-in, first-out) **214**, **706**, 740
 - order **852**
- “light bulb moment” **590**
- `<limits>` header **110**, **132**, **202**
- line **77**
- line number **756**, **759**
- line of text **1109**
- linear regression **420**
- linear relationship **420**, **420**
- linear run time **701**, **1199**
- linear search **1198**, **1203**, **1219**
 - of an array **1201**
- linear search algorithm **702**, **1207**
- link **25**
- linkage **1182**
 - module (C++20) **1182**
 - of an identifier **1183**
- linked list **699**
- linker **26**
- Linux
 - shell prompt **3**, **28**
- Linux operating system **15**
 - kernel **16**
- `list` class **715**, **723**
- `list` class template **704**
- `<list>` header **201**, **723**
- `list` member functions
 - `assign` **727**
 - `merge` **726**
 - `pop_back` **727**
 - `pop_front` **727**
 - `push_front` **725**
 - `remove` **728**
 - `sort` **725**
 - `splice` **725**
 - `swap` **727**
 - `unique` **727**
- literal
 - character **401**
 - digits **401**
 - floating point **156**
- live-code approach **xxiii**
- Live Instructor-Led Training with Paul Deitel **xlvii**
- LL for long long integer literals **132**
- LLVM Clang C++ liv
- LLVM clang++ **3**, **28**
- LLVM Compiler Infrastructure clang++ **xxviii**
- load **25**, **769**
- load a program into memory **349**
- load factor **703**
- load function of `std::atomic` **1051**
- load/store operations **349**
- loader **26**, **27**
- loading phase **354**
- local automatic object **467**
- local variable **109**, **109**, **241**, **482**, **1183**
 - destructors **696**
 - `static` **240**
- `<locale>` header **202**
- locale-specific numeric formatting **1130**
- localize formatted strings **1129**
- location in memory **75**
- location-based services **53**

- lock
 - an object 1021, 1022, 1023
 - release **1019**
- `lock` member function of class `weak_ptr` **1148**, 1151
- `lock_guard` class **1023**
- `log` function 192
- `log10` function 192
- logarithm 192
- logarithmic running time **702**, **1207**
- logging 685
- logging libraries
 - Boost.Log 685
 - Easilylogging++ 685
 - Google Logging Library (`glog`) 685
 - Loguru 685
 - Plog 685
 - spdlog 685
- logic error **26**, **79**, **104**, 148, 151
 - slicing **661**
- `logic_error` exception **681**
- logical AND (`&&`) operator 170, 1185
 - in a constraint 862
- logical complement operator, `!` **171**
- logical function object **822**
- logical negation, `!` **171**
- logical NOT (`!`) 1185
- logical operator keywords 1185
- logical operators **169**
- logical OR (`||`) operator xxix, 169, **170**, 1185
 - in a constraint 862
- logical unit **4**
- `logical_and` function object 822
- `logical_not` function object 822
- `logical_or` function object 822
- Loguru logging library 685
- `long` data type **128**, 128, 199
- `long double` data type **114**, 156, 199
- `long long` (integer) data type **128**, 128, 129, 199, 232
 - LL for literals **132**
- long-running task 1074
- loop
 - body 158
 - continuation condition **98**, 147, 148, 149, 158, 167
 - counter 146
 - infinite **106**
 - nested within a loop 120
 - statement **98**
- loss of data 1126
- `lower_bound` algorithm 837
 - ranges version (C++20) 810, **810**
- `lower_bound` function of associative container **733**
- lowercase letter 11, 73, 90, 201
- “lowest type” 199
- low-level I/O capabilities 1103
 - `lvalue` 174
 - `lvalue` (“left value”) **173**, 220, 261, 313, 314, 468, 610, 630, 729
 - `lvalues` as `rvalues` 173
- M**
 - `m-by-n` array **279**
- machine dependent 13, 331
- machine language **13**
 - programming xxx, 13, 348
- machine learning xxix, xl, **56**, 359, 390, 417, 420
- Machine Learning with Simple Linear Regression
 - Statistics Can Be Deceiving case study exercise xxxi, 417
- Time Series Analysis case study exercise 423
- Machine Learning with Simple Linear Regression exercise xxxi
- Machine Learning with Simple Linear Regression: Time Series Analysis 423
- Macintosh 16
- macOS **16**
 - Terminal window 3
- macro 200, **937**
- magic numbers **270**
- magnitude 1117
- `main` **68**
 - thread **1007**
- “make your point” 209
- `make_heap` algorithm 837
 - ranges version (C++20) **818**
- `make_pair` function **737**
- `make_shared` Standard Library function **1146**
- `make_tuple` function **901**
- `make_unique` function **601**, 603, 621
- malware detection 53
- mangled function name 225
- “manufacturing” section of the computer 6
- `map` associative container 730
- `map` container class template 705
 - `<map>` header 201, **736**, 738
- `map` of Student Grades exercise 751
- mapped values 730
- mapping in functional-style programming **288**, 828
- marketing
 - analytics 53
- mashup **43**
- `match_results` class **405**
 - `suffix` member function of class `match_results` **407**
- Matching Numeric Values exercise 412

- math library functions 191, 201
`ceil` 192
`cos` 192
`exp` 192
`fabs` 192
`floor` 192
`fmod` 192
`log` 192
`log10` 192
`pow` 192
`sin` 192
`sqrt` 192
`tan` 192
 Math Library Functions exercise 255
 mathematical algorithms of the standard library 792
 mathematical constants 193
 mathematical special functions 193
 mathematics xl
`max` algorithm 445, 812, 837
 max heap 816
`max_element` algorithm 837
 ranges version (C++20) 792, 794
`max_size` member function
 container 709
`string` 365
`maximum` function 194
 maximum integer value on a system 993
 maximum size of a string 365, 365
 maximum statistic 418, 751
`mdarray` container (C++23) 283
`mdspan` (C++23) 1192
 mean 77
 measures of central tendency 396, 418
 measures of dispersion 418
 standard deviation 418
 variance 418
 measures of variability 418
 medical device monitoring 53
 megabytes (MB) 50
 member function 23
 call 24
 defined in a class definition 452
 no arguments 452
 parameter 195
 member-initializer list 439, 477, 1171
 member object
 destructors 696
 initializer 478
 member selection operator `(.)` 456, 553, 602
 memberwise assignment 596
 memory 6, 7, 72, 75, 1183
 address 312
`allocate` 598, 598
 consumption 570
`deallocate` 598
 footprint of exceptions 656
 leak 589, 598, 600, 605, 700
 leak, preventing 602
 management 191
 unit 5
 utilization 704
 memory-access violation 700
 memory footprint 769
`<memory>` header 201, 601, 774, 838, 1143
 memory leak 1143
 memory location 75
 memory-space/execution-time trade-off 704
`merge`
 algorithm 837
 algorithm, ranges version (C++20) 802, 803
 member function of associative containers 730
 member function of `list` 726
`merge` member function of associative containers 730
 merge sort algorithm 1213, 1214, 1219
 recursive implementation 1213, 1214
 merge sort algorithm efficiency 1219
 merge symbol in the UML 106
 merge two arrays 1213
 merge two ordered list objects 843
 Merging Ordered Lists 843
 metacharacter (regular expressions) 401
 metafunction 916
 return value 917
 template argument 917
 type 917
 value 917
 metaprogramming xxvii, 18
 Metaverse 44
 Metric Conversions exercise 413
 Meyer, Bertrand
 design by contract 687
 Mi, Ningfang xviii
 microservices 49
 Microsoft 1062
 C++ language documentation xlvi
 Cortana xlvi, 63
 Visual C++ xxviii
 Microsoft Azure 42
 Microsoft modularized standard library 968
 Microsoft open-source C++ standard library 883
 Microsoft Parallel Patterns Library concurrent containers 1062
 Microsoft Visual Studio Code editor 28
 Microsoft Visual Studio Community edition liv, 29
 Microsoft Windows 162
`midpoint` algorithm 838

- mileage obtained by automobiles 135
- milliseconds** object **993**
- min** algorithm 812, **812**, 837
- min heap 816
- min_element** algorithm 837
- ranges version (C++20) 792, **794**
- minimum iterator requirements
- standard library algorithms 774, 776
- minimum statistic **418**, 751
- minmax** algorithm 812, **812**, 837
- ranges version (C++20) **813**
- minmax_element** algorithm 837
- ranges version (C++20) 792, **794**
- minus** function object 822
- mismatch** algorithm 837
- ranges version (C++20) 783, **785**
- mismatch_result** **785**
- mismatch_result** for the **mismatch** algorithm **785**
- missing data **393**, 394
- mixed reality **46**
- mixed-type expression **199**, 133
- mobile application 17
- Modern C++ xxiv, xxvi, xxvii, xxx, xxxii, xxxiv, xxxvii, xxxviii, xli, xlili, xliv, 19, 20, 310
- do more at compile-time 271, 846
- modifiable data **989**
- modifiable *lvalue* 594, 595, 610, 630
- modify a constant pointer 327
- modify address stored in pointer variable 327
- modular exponentiation 526
- modular standard library
- C++23 973
 - Microsoft 967
- module 1182
- module linkage (C++20)
- 1182
- Modules 980
- :private** Module Fragment exercise 979
 - Benefits discussion exercise 978
 - Creating a Module exercise 979
 - Creating a Module exercise modification 979
 - Cyclic Dependencies Are Not Allowed exercise 980
 - Importing Standard Library Headers as Header Units exercise 979
 - Imports Are Not Transitive discussion exercise 979
 - Reduced Translation Unit Sizes exercise 979
 - Separating Interface from Implementation exercise 979
 - Tools discussion exercise 978
 - Visibility vs. Reachability exercise 980
- modules (C++20) xxvii, xxxv, **934**, 951
- building a module with partitions 961, 962
 - clang++** precompiled module (.pcm) file **940**, 948
 - export** a block of declarations 942
 - export** a declaration **942**, 942, 945, 984
 - export** a definition 984
 - export** a namespace 942
- modules (C++20) (cont.)
- export** a namespace member 942
 - export** followed by braces 984
 - export module** **944**
 - export module** declaration 984
 - filename extension .cpp 944
 - filename extension .cppm 944
 - filename extension .ifc 944
 - filename extension .ixx **944**, 944
 - filename extension .pcm 944
 - fmodule-file** compiler flag (clang++) **948**
 - fmodules-ts** compiler flag (g++) **940**
 - fprebuilt-module-path** compiler flag (clang++) **948**
 - global module 984
 - global module fragment 984
 - header unit **938**, 984
 - IFC (.ifc) format 984
 - import** a header file 984
 - import** a module 984
 - import** declaration **946**, 984
 - import** existing header as a header unit **938**
 - improve compilation performance 939
 - interface 942
 - linkage 984
 - Microsoft modularized standard library 968
 - modular standard library (C++23) 973
 - module** declaration **943**, 944, 952, 984

- modules (C++20) (cont.)
 - module implementation
 - partition unit **962**
 - module implementation
 - unit **952**, 984
 - module interface 942
 - module interface partition
 - unit 944, 959, **959**, 959, 960, 961, 962
 - module interface partition
 - unit **export module** declaration 960
 - module interface unit
 - 944**, 984
 - module interface unit
 - compile in **clang++** 948
 - module interface unit
 - compile in **g++** 947
 - module linkage 1182
 - module name 943, 985
 - module partition 985
 - module purview **944**, 985
 - module unit **943**, 985
 - named **952**
 - named module 958, 985
 - named module purview
 - 985
 - partition **958**, 985
 - partition rules 959
 - precompile** compiler flag (**clang++**) **940**
 - precompiled module interface 985
 - primary module interface
 - unit **944**, 960, 985
 - private** module fragment **958**, 985
 - purview **944**
 - reachability **971**
 - reachable declaration 985
 - templates 945
 - visibility **971**
 - visible declaration 985
 - x c++-module** compiler flag (**clang++**) **948**
- modules (C++20) (cont.)
 - x c++-system-header** compiler flag (**g++**) **940**
 - xc++-system-header** compiler flag (**clang++**) **940**
- Modules Project
 - Compilation Performance exercise 980
 - Importing the Standard Library 980
- modulus** function object 822
- modulus operator, % **76**
- monetary calculations
 - cpp_dec_float_50** (Boost Multiprecision open-source library) 180
- monetary formats 202
- Moore's Law **7**, 7
- Moore's law **834**, 991
- most derived class **1177**
- motion information 5
- mouse 4
- move** **611**, 622
- move** algorithm 837
 - ranges version (C++20) **803**
- move** assignment operator xxxiii, 441, 589, **605**, 622, 624, 710
- move** constructor xxxiii, 441, 589, **605**, 611, 612, 622, 623, 708, 710
- move** semantics xxxiii, 589, 609, 710
 - move** assignment operator 612
 - move** constructor 612
 - std::move** function **612**, 612
- move_backward** algorithm 837
 - ranges version (C++20) **803**
- MoveAssignable** 710
- Mozilla Foundation **15**
- multi **991**
- multi-core 775
 - architecture **835**, **991**
 - processor 260, 283
 - systems xxxv
- multi-core processor xliv, **6**, 8
- multicore **835**
- multidimensional array **279**
- multidimensional views of contiguous data 1192
- multiline comment **67**
- multimap** associative container 705, 730, 736
- multipass algorithms **712**
- multiple inheritance **533**, 533, 1169, 1170, 1171, 1172, 1174
 - demonstration 1170
 - diamond inheritance **1174**
- multiple-selection statement **98**
- multiple-source-file program compilation and linking process 454
- Multiples exercise 251
- Multiples of 2 with an Infinite Loop exercise 141
- multiplication **76**
 - compound assignment operator, *= 125
- multiplies** function object 822
- multiset** associative container 705, 730
- multithreading xliv, 8, **989**, 989
 - condition variable 1021
 - launch** enum **1046**
 - std::async** function template **1046**
 - std::future** class template 1046
 - std::packaged_task** function template **1047**
 - std::shared_future** class template 1047

- multithreading and multicore systems performance case studies xxxv, xxxvi
- multivariate time series **423**
- Munging Dates exercise 412
- mutable**
 - data member **1184**, 1184
 - demonstration 1184
 - keyword 1182, **1184**
- mutable (modifiable) data **989**, 1016
- mutable data **989**
- mutating sequence algorithms **836**, **836**
- mutex** class **1019**, 1020
- <**mutex**> header 202, **1019**, 1036
- mutual exclusion **1016**, 1019, 1020, 1059
 - necessary condition for deadlock 1002
 - thread safety 989
- Mutual Exclusion discussion exercise 1066
- MyArray** class 604, 606, 883, 893
 - definition 615
 - definition with overloaded operators 615
 - test program 606
- MyArray** Class Template exercise 930
- MyArray** Container Class
 - Template with Custom Random-Access Iterators
 - project exercise 931
- mystery recursive exercise 238, 239
- N**
 - name handle 456
 - on an object 456
 - name mangling **225**
 - to enable type-safe linkage 225
 - name of a variable **75**, 1182
 - name of an array **261**
- named entity recognition **425**
- named module (C++20 modules) **952**, 958
- named requirements **710**
- named return value optimization (NRVO) **611**, 632
- namespace** 1179
 - alias **1182**
 - global **1180**
 - keyword **945**, 1180
 - member **946**
 - nested **1180**
 - qualifier 946, 1181
 - scope **239**
 - std** **69**
 - std::chrono** 993
 - unnamed **1180**
- naming conflict 482, 945
- NaN (not a number) 394
- narrow_cast** operator **199**
- narrowing conversion **123**, 198, 199
 - braced initializer 198
 - explicit 199
- National Oceanic and Atmospheric Administration (NOAA) 423
- Natural** **425**
- natural language **424**
- natural language processing (NLP) xxxi, xliv, 53, 359, 415, **425**, 426
- natural logarithm 192
- Navigator** area (Xcode) 38
- Navigators
 - Issue 38**
 - Project 38**
- NDEBUG to disable assertions 671
- near container **706**
- negate** function object 822
- negative infinity (-inf) 658
- nested
 - blocks 239
 - building block 179
 - control statements **118**, 121, 177, 179
- nested (cont.)
 - for statement 273, 282
 - if...else selection statement 139
 - if...else statement **102**
- namespace** **1180**, 1180
- parentheses **77**
- placeholder in a format specifier 1132
- requirement in C++20 concepts 874, **876**
- try blocks 667
- type** **706**
- type names in containers 892
- type names in iterators 887
- nested_exception** 665
- nesting rule **177**
- network connection 1102
- network message arrival 669
- neural network 56
- new**
 - failure 676
 - failure handler **679**
 - operator **598**
- <**new**> header 676
- new** operator
 - calls the constructor 598
 - placement version 598
 - returning **nullptr** on failure 678
 - throwing **bad_alloc** on failure 677, 678
- newline ('\n') escape sequence **69**, 69, 340, 1106
- next_permutation** algorithm 838
- NeXTSTEP operating system 16
- n-grams **426**
- NLP (natural language processing) 415
- NLP (natural language processing) 53
- [[nodiscard]] attribute 457, **1188**

- no guarantee (of what happens when an exception occurs) **664**
- no preemption (necessary condition for deadlock) **1002**
- no throw exception safety guarantee **664**
- `noboolalpha` stream manipulator **1122**
- `node_type` in an associative container **709**
- NodeJS **22**
- `noexcept` **623**
- `noexcept` keyword **623**, 664, 675, 688
- `non-const` member function **474**
- on a `const` object **473**
 - on a `non-const` object **473**
- non-type template parameter **892**
- nonconstant pointer to constant data **326**
- nonconstant pointer to non-constant data **326**
- noncontiguous memory layout of a deque **728**
- nondeterministic random numbers **204**
- `none_of` algorithm **837**
- ranges version (C++20) **796, 799**
- nonfatal logic error **104**
- nonfatal runtime error **28**
- non-fungible token (NFT) **47**
- non-member function to overload an operator **635**
- nonmodifiable *lvalue* **594**, 595
- nonmodifying sequence algorithms **836, 837**
- non-module translation unit **951**
- non-parameterized stream manipulator **117**
- nonrecoverable failures **1126**
- `non-static` member function **482, 640**
- non-virtual interface idiom (NVI) **1142, 1161**
- nonzero treated as `true` **173**
- `noshowbase` stream manipulator **1111, 1119**
- `noshowpoint` stream manipulator **1116**
- `noshowpos` stream manipulator **1111, 1118**
- `noskipws` stream manipulator **1111**
- not a number **164**
- not equal **79**
- `not` operator keyword **1185**
- `not_eq` operator keyword **1185**
- `not_equal_to` function object **822**
- note in the UML **98**
- `nothrow` object **678**
- `nothrow_t` type **678**
- `notify_all` function of a `std::condition_variable_any` **1037**
- `notify_one` function of a `std::condition_variable` **1021**, 1022, 1023, 1031
- noun phrase extraction **425**
- `nouppercase` stream manipulator **1111, 1121**
- NRVO (named return value optimization) **611**, 632
- `nth_element` algorithm **838**
- `NULL` **312**
- null character ('\0') **340**, 1111
- `null` in JSON **516**
- null pointer (0) **312**, 314
- null-terminated string **341**, 1105
- `nullptr` **312**, 670
- on new failure **678**
- number of arguments **195**
- number systems appendix on deitel.com **xli**
- numbers
- format with their signs (+) **1131**
- `<numbers>` header **193**
- numbers in JSON **516**
- numeric algorithms **822, 838**
- numeric formatting
- locale specific **1130**
- `<numeric>` header **284**, 774, 813, 838
- `numeric_limits` class template **110, 132**
- `max` function **993**
- numerical data **414, 415**
- numerical data type limits **202**
- NVI (non-virtual interface idiom) **1161**

O

- `o` presentation type **1130**
- $O(1)$ **700, 1199**
- $O(\log n)$ **702**, 1207
- $O(n \log n)$ **1219**
- $O(n)$ **701**, 1199, 1202, 1210, 1219
- $O(n^2)$ **701**, 1200, 1213
- object
- leaves scope **463**
 - lifetime **436**
 - of a class **23, 24**
 - of a derived class **546, 549**
 - of a derived class is instantiated **544**
- object code **26**
- object oriented
- analysis and design (OOAD) **25**
 - programming (OOP) **16**, 18, 25
- object-oriented programming (OOP) **xxvii, 18, 530**
- object's *vtable* pointer **573**
- Objective-C **16**
- objects contain only data **454**

- Objects-Natural Approach 4, 66
 Objects Natural Case Study
 Super-Sized Integers 129
 objects-natural case study
 xxvii, xxviii, xxix, xxx, xxxi, xxxii, 83
 Creating and Using Objects of Standard-Library Class string 83
 Precise Monetary Calculations with the Boost Multiprecision Library 180
 observations in a time series
 xxxi, **423**
 oct stream manipulator
 1111, **1112**, **1119**
 octa-core processor **8**
 octal (base 8) number system
 1130
 octal (base-8) number system
 1111, 1112
 octal number 1105, 1119
 odd integer 184
 O'Dwyer, Arthur xlix
 off-by-one error **148**
 offset
 from the beginning of a file **383**
 into a vtable **572**
`ofstream` class 378, 379, 380
`once_flag` **1048**
 One Definition Rule (ODR) **936**, 937
 one-pass algorithm 712
 one-time, thread-safe initialization of an object 1047
 one-to-many
 mapping 705, 706
 relationship **736**
 one-to-one mapping 705, 706, 738
 online forums xlvi
 OOAD (object-oriented analysis and design) 25
 OOP (object-oriented programming) 18, **25**, 530
 open a file
 for input 380
 for output 380
 that does not exist 380
 open source **15**, 17
 code xxxviii
 community xxxix
 increases productivity **15**
 Microsoft C++ standard library 883
 movement xxiv
 software xliv
 Open Web Application Security Project (OWASP) 516
 OpenAI 16, 57
 OpenCV **16**
 opened 377
 OpenML **16**
 open-source libraries
 Boost Multiprecision xxix, 180
 operand **69**, 74, 349, 760
 operating system xxiii, xliv, **14**, 16, 18
 device driver polymorphism 559
 operation code **349**, 760
 operator
 `*` (pointer dereference or indirection) **314**, 315
 `+=` (addition assignment)
 361
 address (`&`) 315
 arrow member selection (`->`) **456**
 `co_await` **1089**
 `delete` **598**
 member selection `(.)` 456
 `narrow_cast` **199**
 `new` **598**
 overloading 226
 precedence and grouping chart 127
 scope resolution `(::)` 451
 `sizeof` **329**, 329
 operator (cont.)
 `sizeof...` **894**
 `static_cast` **644**
 that cannot be overloaded 596
 that you do not have to overload 596
 unary scope resolution `(::)` **244**
 operator
 functions **596**
 keyword 596, **596**
 operator `bool` member
 function 380, 383, 1127
 operator keywords **1185**, 1186
 demonstration 1186
 operator overloading 132, 292, **588**
 addition assignment operator `(+=)` 614
 addition operator `(+)` 596, 597
 binary operators 597
 cast operator **630**
 choosing member vs. non-member functions 635
 commutative operators **635**
 constructors 1169
 conversion operator **630**
 copy assignment `(=)` **592**, 609, 621
 decrement operators 631
 equality operator `(==)` 609, 627
 function call operator `()` 644, **644**, 1192
 in templates 929
 increment operators 631
 inequality operator 608, 628
 is not automatic 596
 member vs. non-member functions 635
 `operator[]` 629
 `operator+` 596

operator overloading (cont.)
operator++ 631, 632
operator<< 634
operator= 621
operator== 627
operator>> 634
 postincrement operator
 $(++)$ 632
 preincrement operator
 $(++)$ 631, 632
 rules and restrictions 597
 self-assignment **594**
 stream extraction operator
 $>>$ 634
 stream insertion and
 stream extraction operators 606, 607, 614
 subscript operator 610, 629
operator! member function
 1127
operator[]
 const version 629
 non-const version 629
operator+ 596
operator<< 634
operator= 621, 708
operator== 627, 784
operator>> 634
 operators xxix
 $--$ (predecrement/post-decrement) **125**, 126
 $!$ (logical negation) 169, **171**
 $!=$ (inequality) 79
 $?:$ (ternary conditional) 104
 $()$ (parentheses) 77
 $*$ (multiplication) 76
 $*=$ (multiplication assignment) 125
 $/$ (division) 76
 $/=$ (division assignment) 125
 $&&$ (logical AND) 170
 $\%$ (remainder) 76
 $\%=$ (remainder assignment) 125

operators (cont.)
 $+$ (addition) 74, 76
 $++$ (prefix increment/postfix increment) 126
 $++$ (preincrement/postincrement) 125
 $+ =$ (addition assignment) **125**
 $<$ (less-than operator) 79
 $<<$ (stream insertion) **68**, 74
 $<=$ (less-than-or-equal-to) 79
 $=$ (assignment) 74, 76
 $- =$ (subtraction assignment) 125
 $==$ (equality) 79
 $>$ (greater-than) 79
 $>=$ (greater-than-or-equal-to) 79
 $>>$ (stream extraction) **74**
 $\|$ (logical OR) 169, **170**
 associativity **77**
 compound assignment
 125, 127
 conditional operator, $:?$ **104**
decltype 1186
 decrement operator, $--$
 125, 126
 dot $(.)$ **85**
 grouping **77**
 logical AND, $\&\&$ 170
 logical complement, $!$ **171**
 logical negation, $!$ **171**
 logical operators **169**, 171
 logical OR, $\|$ 169, **170**
 modulus, $\%$ **76**
 overloading **75**
 postfix decrement **126**
 postfix increment **126**
 precedence 77
 precedence and grouping chart 82
 prefix decrement **126**
 prefix increment **126**
 remainder, $\%$ 76, **76**, 140, 251, 255

Optimal Size of a Circular Buffer discussion exercise 1069
 optimizing
 code 770
 compiler 156
 Simple compiler 769
optional class 311
<optional> header 202
or operator keyword **1185**
or_eq operator keyword **1185**
 order in which actions should execute **95**
 order in which constructors and destructors are called 464, 466, 544
 order of evaluation of operators 89
 Order of Exception Handlers 696
 ordered associative container 704, **705**, 705, 730
 ordinary least squares **420**
 O'Reilly Online Learning
 free trial li
 original format settings 1123
 OS X 16
ostream class 383, **1103**
 seekp function **383**
 tellp function **383**
ostream_iterator **710**
ostringstream class **386**, 386, 626
 out-of-bounds array elements 264
 out of scope 243
out_of_bounds exception 604, 721
out_of_range exception **295**, 360, 373, **682**
 header **<stdexcept>** 630
 outer block 240
 outer **for** statement 282
 outliers 400

- output xxviii
 buffering 1127
 char * variables 1105
 characters 1104
 data items of built-in type 1104
 floating-point value 1105, 1111
 format of floating-point numbers 1120
 integers 1105
 standard data types 1104
 uppercase letters 1105
 output device 5
 output iterator 712, 713, 714, 778
 output sequence 710
 output stream 720
 output to string in memory 202
 output unit 5
output_iterator concept (C++20) 777, 782
output_range concept (C++20) 777, 782
 outputting to **strings** in memory 386
overflow_error exception 681
 overhead of runtime polymorphism 570
 overload set in overload resolution 857, 857, 878
 overloaded function definitions 224
 overloaded parentheses operator 730
 overloaded stream insertion operator << 634
 overloading 75, 224
 concept based 872
 constructor 462
 function templates 871
 functions 871
 resolution 871
 overloading << and >> 226
 overloading operators 226
- overload-resolution rules 629
override 553, 556
 override a function 552, 552
override keyword 553, 556
 OWASP (Open Web Application Security Project) 516
- P**
- P operation on Dijkstra semaphore 1059
 pack a **tuple** 901, 901
 packets (on the Internet) 41
 pad with specified characters 1105
 padding characters 1111, 1114, 1117, 1118
pair 733
pair class template 899
 pair of braces {} 82
 palindrome exercise 140, 307, 409, 843
palindrome function 751
 Palindromes exercise 751
par execution policy of a parallel algorithm 994, 995
par_unseq execution policy of a parallel algorithm 995
 parallel algorithms xxxv, 835
std::execution::par execution policy 994, 995
std::execution::par_unseq execution policy 995
std::execution::parallel_policy class 995
std::execution::parallel_sequenced_policy class 995
std::execution::seq execution policy 995
std::execution::sequenced_policy class 995
- parallel algorithms (cont.)
std::execution::unseq execution policy 995
std::execution::unsequenced_policy class 995
 parallel execution 996
 parallel operations 988, 988
parallel_policy class 995
parallel_sequenced_policy class 995
 Parallelizing 60,000,000 Die Rolls exercise 1070
 parameter 195
 list 195
 parameter pack 894, 903
 expansion 904
 variadic template 905
 parameterized stream manipulator 117
 quoted 384
 parameterized type 849
 parentheses operator () 77
 parentheses to force order of evaluation 82
 Parking Charges exercise 249
 partial template specialization 924, 924
partial_sort algorithm 838
partial_sort_copy algorithm 838
partial_sum algorithm 813, 815, 838
 partition 985
partition algorithm 838
 partition in a C++20 module 958
 name 959
 partition step in quicksort 1223
partition_copy algorithm 838
partition_point algorithm 838

- partitions (C++20 modules)
 building a module with 961, 962
 rules 959
- parts-of-speech (POS) tagging **425**
- pass-by-pointer 316
- pass-by-reference **219**, 311, 316, 317, 319
 with a pointer parameter used to cube a variable's value 317
 with pointer parameters **316**
 with reference parameters 219, 316
- pass-by-value **219**, 219, 316, 318, 326
 vs. pass-by-reference 219
 vs. pass-by-reference exercise 256
- Password Format Validator exercise 412
- path 380
- pattern of 1s and 0s 10
- Paul Deitel
 Live Instructor-Led Training xlvi
- Payroll System Modification 584
 exercise 584, 585
- payroll system using runtime polymorphism 560
- .pcm (precompiled module) file in clang++ **940**, 944, 948
- Pearson eText xlvi
- Pearson Instructor Resource Center xlvi
- Pearson Revel xlvi
- peek function of *istream* **1109**
- percent sign (%) (remainder operator) **76**
- perfect number 251
 exercise 251
- perform operations sequentially 988
- performance 20
- performance issues with exceptions 661, 670
- performance tips xxvii
- performance-intensive systems xliv
- performing operations concurrently **988**
- permutable concept (C++20) 793
- persistent storage **6**
- personal assistants
 Amazon Echo 17
 Apple HomePod 17
 Google Home 17
- personalized medicine 53
- personally identifiable information (PII) 63
- petabyte (PB) 50
- petaflops 9
- Peter Minuit exercise 187
- phishing elimination 53
- Pi (π) 89
- Pig Latin exercise 409
- pipeline in C++20 ranges 288
- placeholder (in text formatting) **153**, 155
- placeholder type 1094
- placement *delete* 598
- placement *new* 598
- plaintext 522
- plaintext in cryptography 497
- platform dependency 1001
- Plog logging library 685
- plus function object 822
- plus sign 1118
- pointer 310, 311, 331
 arithmetic **331**
 arithmetic, machine dependent 331
 comparison 334
 declared *const* 327
 dereference (*) operator **314**, 315
 expression 331
- pointer (cont.)
 handle 456
 operators & and * 315
 to a function **570**, 570
 to an implementation 575
 to dynamically allocated storage 620
 to *void (void *)* **333**
- pointer 707
- pointer-based array xxx, 310, 311
- pointer-based string 310, xxx, 310, 311, 340
- pointer nested type in an iterator 887
- pointers xxx
- poker playing program 515
- poll analysis program 276
- pollution reduction 53
- Polymorphic Banking Program Exercise Using Account hierarchy 584
- polymorphic behavior 553
- polymorphic processing 344
- polymorphic video game 545
- polymorphically invoking functions in a derived class 1174
- polymorphism 165, 508, 573
 compile-time (static) 710, **847**, 849
 runtime 531
- Polymorphism and Extensibility 582
- Polynomial class 651
- pop
 member function of a stack 852
 member function of container adapters **739**, **740**
- member function of *pri*ority_ *queue* 743
- member function of *queue* 742
- member function of *stack* 740

- pop off a stack **214**
- pop_back member function
 - of `list` **727**
- pop_front **724, 729, 742**
- pop_heap algorithm **837**
 - ranges version (C++20) **819**
- portability **94, 128**
- portable **18**
- position number **261**
- positional argument in a format string **1129**
- positional value **141**
- positive infinity (`inf`) **658**
- post contract keyword
 - (GNU C++ early access implementation) **689**
- postcondition **686**
 - violations **686**
- postfix decrement operator **126**
- postfix evaluation **761**
- Postfix Expression Evaluator
 - exercise **755**
- postfix increment operator **126**
- postfix notation **754, 761**
- postfix-expression evaluation
 - algorithm **754**
- postincrement **126, 126, 127**
- postincrement an iterator **713**
- pow function **156, 192**
- power **192**
- power of 2 larger than 100
 - 106**
- #pragma once **448, 450**
- pre contract keyword (GNU C++ early access implementation) **689**
- precedence **77, 78**
 - of arithmetic operators **xxviii**
- precedence and grouping
 - chart **127**
- precedence chart **82**
- precedence not changed by overloading **597**
- precise monetary calculations
 - 180**
 - `cpp_dec_float_50`
 - (Boost Multiprecision open-source library) **180**
- precision **117, 1105, 1111**
 - floating-point numbers **1112**
 - floating-point value **114, 155**
 - setting **1113**
- precision function of `ios_base` **1113**
- precision medicine **53**
- precompile compiler flag (clang++) **940**
- precompile a header as a header unit **940**
- precompiled module (.pcm) file (clang++) **940, 948**
- precondition **685**
 - violations **686**
- prederection **126**
- predefined function objects **822**
- predicate function **456, 725, 785, 789, 791, 797, 798, 799, 803, 807, 810**
- predicted value in simple linear regression **420**
- preemptive scheduling **1001**
- prefix decrement operator **126**
- prefix increment operator **126**
- preincrement **126, 126, 127**
- preincrement operator **(++)**
 - overloaded **631, 632**
- "prepackaged" functions **191**
- preprocessing directives **26**
- preprocessor xli, **25, 26**
 - directives **67**
 - state **939**
- presentation type (C++20 text formatting) **1128**
 - a (or A) **1129**
 - b (or B) **1130**
 - c **1130**
 - d **1130**
- presentation type (C++20 text formatting) (cont.)
 - e (or E) **1129**
 - f **1129**
 - g (or G) **1129**
 - o **1130**
 - x (or X) **1130**
- presentation type (string formatting)
 - c **409**
- prev_permutation algorithm **838**
- prevent memory leak **602**
- preventative medicine **53**
- primary memory **6, 26**
- primary module interface unit (C++20 modules) **944, 960**
- prime number **252, 526, 751**
 - exercise **252**
 - factorization **751, 1040**
- University of Tennessee Martin Prime Pages website **1041**
- principal **187**
 - in an interest calculation **153**
- principle of least privilege **240, 326, 447, 472, 713**
- print a string backward recursively exercise **308**
- print an array recursively exercise **307**
- print function (C++23) **1190**
- print spooling **1008**
- printer **27, 1102**
- printing
 - Decimal Equivalent of a Binary Number exercise **140**
 - line of text with multiple statements **70**
 - multiple lines of text with a single statement **70**
 - string Backward exercise **409**

- `println` function (C++23)
 - 1190
- `priority_emplace` member function
 - of `queue` 743
- `priority_queue` adapter class **743**, 816, 817, 818
 - `emplace` function 743
 - `empty` function 743
 - `pop` function 743
 - `push` function 743, 750
 - `size` function 743
 - `top` function **743**
- `priority_queue` container class template 706
- privacy xxix, xl, xlv, 63, 64, 129, 246
- private**
 - access specifier **437**, 437
 - base class 1178
 - base-class data cannot be accessed from derived class 539
 - inheritance 534, **1177**
 - members of a base class 534
 - `static` data member 488
- private decryption key **521**
- private key **521**, 522, 524
- private libraries 26
- `:private` module fragment (C++20 modules) **958**
- `private virtual` function 1162
- probability 204
- problem solving xxix, xlii
- procedural programming xxvii, 18
- procedure for solving a problem 95
- Processing Asynchronous Tasks in Separate Threads with Coroutines exercise 1099
- processing unit 4
- producer 989, **1008**, 1009
- Producer/Consumer Relationship discussion exercises 1069
- producer-consumer relationship **1008**
- Product of Odd Integers exercise 184
- production (Simple compiler) **769**
- profiling xxxv, 495, 991, 996
- Profiling Sequential and Parallel Algorithms exercise 1069
- program **4**, 4
- program control **95**
- program development environment 25
- program development tool 114
- program execution stack **214**
- program in the general 531, 582
- program in the specific 531
- program termination 467
- program to an interface, not an implementation 573
- programmer **4**
- programming xl
- programming fundamentals xxiv, xl
- Programming in the General exercise 582
- programming languages xxviii
 - BASIC 21
 - C 18
 - C# 21
 - Go 22
 - Java 21
 - JavaScript 22
 - Kotlin 21
 - Python 21
 - Rust 22
 - Swift 22
 - Visual C++ 21
- programming paradigms
 - functional-style xxvii, 18
 - generic xxvii, 18
 - metaprogramming xxvii, 18
 - object-oriented xxvii, 18
 - procedural xxvii, 18
- programming tips
 - C++ Core Guidelines xxvii, xli
 - C++20 modules xxvii
 - common programming errors xxvii
 - performance tips xxvii
 - security best practices xxvii
 - software engineering observations xxvii
- Programming to an Interface Modification exercise 585
- project 29, 38
- Project Gutenberg xlv, 426
- Project** navigator 38
- projection in C++20
 - `std::ranges` algorithms 785, 825
- promise object (coroutines) **1094**
 - `final_suspend` member function **1095**
 - `get_return_object` member function **1094**
 - `initial_suspend` member function **1094**
 - `return_value` member function **1094**
 - `return_void` member function **1094**
 - `unhandled_exception` member function **1094**
 - `yield_value` member function **1095**
- promote `int` to `double` **117**
- promotion **117**
- promotion rules **198**
- prompt **74**
- prompting message 1127

- property injection **576**
 proprietary software **15**
protected **1160**
 - base class **1178**
 - base class member function **1161**
 - data, avoid **1161**
 - inheritance **534, 1177, 1178**
 - member of a class **1160**
 - virtual** function **1162****protected** data (avoid) **1161**
 protecting the environment **53**
pseudocode **95**, 101
 - first refinement **112**, 119
 - second refinement **112, 120****pseudorandom numbers** **207**
-pthread compiler flag **1003**
public
 - member of a subclass **1160**
 - method **451****public** access specifier **437**
public base class **1178**
public encryption key **521, 523**
public inheritance **534, 1177**
public key **521**, 522, 524
public keyword **437**
public member of a derived class **534**
public services of a class **448**
public static
 - class member **488**
 - member function **488****public-key cryptography** xxix, xxxii, xliv, **506, 521**
public-key/private-key pair **522**
pure abstract class **559**, 574
pure specifier ($= 0$) for a **virtual** function **559**
pure virtual function **559**
purview (C++20 modules) **944**
- push** member function
 - container adapters **739, 740**
 - priority_queue** **743**
 - queue** **742**
 - stack** **740, 852****push** member function of **priority_queue** **750**
push onto a **stack** **214**
push_back member function **vector** **295**, 717
push_front member function
 - deque** **728**
 - list** **725****push_heap** algorithm **837**
 - ranges version (C++20) **819****put** member function **1105, 1106**
put pointer **383**
putback function of **istream** **1109**
 Pythagorean Triples exercise **186**
 Python programming language **21**
 Python Software Foundation **16**
- Q**
- QR code **5**
quad-core processor **8**
quadratic run time **1200**
quadratic running time **701**
Quadrilateral Inheritance Hierarchy exercise **583**
Quality Points for Numeric Grades **252**
qualityPoints **252**
quantifier
 - ? **403**
 - {*n*,} **404**
 - {*n, m*} **404**
 - * **402**
 - + **403**
 - greedy **403**
 - in regular expressions **402****quantum** **1000**
- quantum computers **9**
 questions: getting answered **xlvii**
queue **699**
queue adapter class **742**
 - back** function **742**
 - emplace** function **742**
 - empty** function **742**
 - front** function **742**
 - pop** function **742**
 - push** function **742**
 - size** function **742****queue** container class template **706**
<**queue**> header **201, 742, 743**
quicksort algorithm **1219, 1223**
quotation marks **68**
quoted stream manipulator **384**
- R**
- race condition **1015**
radians **192**
radius of a circle **141**
RAII (Resource Acquisition Is Initialization) **589, 600, 605, 671, 676, 683, 1008, 1022**
raise to a power **192**
RAM (Random Access Memory) **6**
random-access iterator **712, 728**
 - operations **714, 884****random access to elements of a container** **704**
<**random**> header **201, 203**
random integers in range 1 to 6 **204**
random number **207**
 - generation xxix
 - generation to create sentences **300****random-number generation distribution** **204**
engine **204**

- Random Sentences exercise 300
random_access_iterator concept (C++20) 777, 787, 788, 793, 797, 818, 873
random_access_range concept (C++20) 777, 793, 797, 818, 819
random_device random-number source 208, 213
 randomizing 207
 die-rolling program 207
 Random-Number Simulation case study xxix, xxx
 exercise xxxii
range (C++20) 286, 699, 710
range adaptor (C++20) 829
 `all` 829
 `common` 829
 `counted` 829
 `drop` 829, 832
 `drop_while` 829, 832
 `elements` 829, 834
 `filter` 829
 `keys` 829, 833
 `reverse` 829, 831
 `split` 829
 `take` 829, 831
 `take_while` 829, 831
 `transform` 829, 832
 `values` 829, 833
range-based algorithms (C++20) xxxiv
range-based for statement 267, 360
 with initializer 268
range checking 111, 360, 604
range concept (C++20) 777
range factory (C++20) 830
 `iota` for an infinite range 830, 830
range of elements 722
range statistic 418, 751
range-v3 project 839
range variable 268, 281
ranges (C++20) xxx, xxxiv
 ranges concepts (C++20) 777
 `<ranges>` header (C++20) 203, 286
 ranges library (C++20) 286, 393
 rapidcsv header-only library 390
 `Document` class 391
 `GetColumn` member function of class `Document` 391, 394
 `GetRowCount` member function of class `Document` 395
 rapidcsv library 359
RapidJSON library 390
ratio of successive Fibonacci numbers 234
RationalNumber class 651
raw data 385
raw string literal 389
Raz, Saar xl ix
rbegin
 member function of containers 708
 member function of `vector` 719
Rdatasets repository 414
rdstate function of `ios_base` 1126
reachability (C++20 modules) 971
read 761, 885
read and print a sequential file 382
read data sequentially from a file 381
read member function of `istream` 1110, 1110
readability 120, 425
Readers and Writers discussion exercise 1069
readers and writers problem 1036
 `std::shared_mutex` class 1036
ready thread state 1000
real-time systems xxiii, xliv, 18
real-world data xl
 “receiving” section of the computer 5
Recognizing Palindromes exercise 409
recommender systems 53
record 11, 378
recover from errors 1126
Rectangle Class exercise 512
recursion xxix, 229, 254
 binary search 239
 binary search exercise 1223
 determine whether a string is a palindrome exercise 307
 Eight Queens exercise 307
 examples and exercises 238
 factorial 231
 find the minimum value in an `array` exercise 308
 linear search exercise 1223
 overhead 238
 print a string backward exercise 308
 print an `array` exercise 307
 recursion step 230, 236
 recursive call 230, 235, 236
 recursive solution 238
 recursively generating Fibonacci numbers 236
 step 1223
 visualizing 231
 vs. iteration 237
recursive
 call 231
 check if a string is a palindrome 239
 Eight Queens exercise 239
 Exponentiation exercise 253

- recursive (cont.)
 - factorial function 238
 - Fibonacci function 238
 - function **229**
 - greatest common divisor 238, 254
 - linear search 239
 - mergesort 239
 - minimum value in an array 239
 - print a string backward 239
 - print an array 239
 - quicksort 239
 - raise an integer to an integer power 238
 - Towers of Hanoi 238
- recursive binary search 1219, 1223
- recursive implementation of merge sort 1214
- recursive linear search **1201**, 1219, 1223
- recursively determine whether a string is a palindrome exercise 843
- reddit
 - reddit.com/r/cpp/ xlvi
- reduce
 - algorithm 813, **815**, 838
 - parallel algorithm **998**
- reducing carbon emissions 53
- reducing energy use 53
- reduction **271**, 284, 289, 815, 905
- redundant parentheses **78**
- refactor 531, 575
 - payroll example 574
- refactoring **48**
- Refactoring a Class Hierarchy to Use Interface Inheritance and Dependency Injection project exercise 585
- reference 707
 - argument 316
 - parameter 219, **219**
 - to a constant 221
- reference (cont.)
 - to a local variable 221
 - to a private data member 468
 - to an int 219
- reference count **1143**
- reference nested type in an iterator 887
- refinement process 112
- <regex> header **400**, 405
- regex library
 - cmatch** **405**
 - match_results** class **405**
 - regex_constants** **406**, 406
 - regex_match** function **401**
 - regex_replace** function **405**
 - regex_search** algorithm **405**
 - regex_search** function **405**
 - smatch** **405**
- regression line xxxi, 419, **420**
- regular expression 191, 386, **399**, 406
 - ? quantifier **403**
 - [] character class **402**
 - {n,} quantifier **404**
 - {n,m} quantifier **404**
 - * quantifier **402**
 - \ metacharacter **401**
 - \d character class 401
 - \D character class **401**
 - \d character class **402**
 - \S character class **401**
 - \s character class **401**
 - \W character class **401**
 - \w character class **401**
 - + quantifier **403**
 - caret (^) metacharacter **402**
 - case insensitive 401
 - case sensitive 401
- regular expression (cont.)
 - character class **401**, 402
 - escape sequence **402**
 - metacharacter **401**
 - regex_constants::icase** 406
 - search pattern 399
 - validating data 399
- Regular Expression exercises
 - Capturing Substrings 412
 - Condense Spaces to a Single Space 412
 - Counting Characters and Words 412
 - Matching Numeric Values 412
 - Munging Dates 412
 - Password Format Validator 412
 - Testing Regular Expressions Online 412
- reinforcement learning 56, **57**
- reinventing the wheel 180
- relational
 - function object **822**
 - operator **79**, 80
- relational database **11**
- release
 - a lock **1019**, 1022
 - a semaphore 1059
- release member function of **std::binary_semaphore** **1061**, 1062
- relinquish the processor (**yield**) 1050
- remainder after integer division 76
- remainder compound assignment operator, %= 125
- remainder operator (%) 76, **76**, 77, 90
- remainder operator, % 140, 251, 255
- remove algorithm 837
- ranges version (C++20) 786, **787**

- `remove` member function of `list` 728
- `remove_copy` algorithm 837
 - ranges version (C++20) 786, 788
- `remove_copy_if` algorithm 837
 - ranges version (C++20) 786, 790
- `remove_if` algorithm 837
 - ranges version (C++20) 786, 789
- `remove_prefix` member function of `string_view` 376
- `remove_suffix` member function of `string_view` 376
- Removing break and continue exercise 187
- `rend`
 - member function of containers 709
 - member function of `vector` 719
- repetition
 - counter controlled 116, 119, 120
 - definite 107
 - sentinel controlled 111, 113, 115, 116
- repetition statement 98, 113
 - `do...while` 98
 - `for` 98
 - `while` 98, 109, 116
- `replace ==` operator with = 173
- `replace` algorithm 790, 837
 - ranges version (C++20) 790, 790
- `replace` member function of class `string` 370, 371
- `replace_copy` algorithm 837
 - ranges version (C++20) 790, 791
- `replace_copy_if` algorithm 837
 - ranges version (C++20) 790, 792
- `replace_if` algorithm 837
 - ranges version (C++20) 790, 791
- Replacing `continue` with a Structured Equivalent exercise 187
- representational error 157, 180
 - in floating point 157
- `request_stop` member function of a `std::jthread` 1039, 1040
- requirements 25
- `requires` clause (C++20) 860
- `requires` expression (C++20) 874
- research and project exercises xliv
- reserved word 98
 - `false` 100, 101
 - `true` 100
- `reset` 744
- `reset` member function of class `shared_ptr` 1147
- `resize` member function of class `string` 367
- Resource Acquisition Is Initialization (RAII) 589, 600, 671, 676, 683
- resource leak 602, 662, 683
- resource sharing 1001
- restore a stream's state to "good" 1126
- `result` (`concurrency`) 1081
- rethrow an exception 665
- Rethrowing Exceptions 696
- return a value 68
- Return* key 74
- return key 353
- `return` statement 69, 70, 196
- `return_value` function of a coroutine promise object 1094
- `return_void` function of a coroutine promise object 1094
- returning a reference from a function 221
- returning a reference to a `private` data member 468
- Returning Error Indicators from Class `Time`'s *set* Functions exercise 512
- reusable software components 23
- reuse 24, 85, 434
- Revel (Pearson) xlvi
- `reverse` algorithm 837
 - ranges version (C++20) 802, 804
- Reverse Digits exercise 252
- `reverse` range adaptor (C++20) 829, 831
- `reverse_copy` algorithm 837
 - ranges version (C++20) 805, 807
- `reverse_iterator` 707, 708, 713, 719
- Reversing a Sentence exercise 409
- `rfind` member function of class `string` 368
- Richer Shape Hierarchy 583
- ride sharing 53
- right align 1117
 - > (C++20 text formatting) 155, 156, 1130, 1131
- right brace {} 68, 109, 116
- right fold
 - binary 909
 - unary 908
- right operand 69
- right shift operator (>>) 588

- r**
right stream manipulator 1111, **1117**
right triangle 141
right value 173
right-align text 1131
rightmost (trailing) arguments 223
rise-and-shine algorithm 95
Ritchie, Dennis 18
robo advisers 53
robust application 655
rolling dice 205, 209
Romeo and Juliet xlvi
rotate algorithm 498, 837
rotate_copy algorithm 837
round a floating-point number for display purposes 117
round-robin scheduling **1001**
rounding
 fair rounding algorithm 180
rounding numbers 110, **117**, 157, 192, 1138
Rounding Numbers exercise 249, 250
rows **279**
Roy, Patrice xlvi
RSA Public-Key Cryptography algorithm xxix, xxxii, **129**, **521**, **1040**
 ciphertext cracking 527
 problem 527
Rule of Five (for special member functions) 619
 Rule of Five defaults **619**
Rule of Zero (for special member functions) 442, **619**, 885
rules for forming structured programs 176
rules of operator precedence **77**
running state 1000
running total 112
runtime (concurrentpp) **1081**, 1083
runtime error **28**
runtime polymorphism **531**
 using class hierarchies 1161
 with **virtual** functions 554
runtime_error class **658**, 668, 681
 what function 663
runtime-polymorphism case study xxxii
Rust programming language 22
rvalue ("right value") **173**, 174, 221
rvalue reference (&&) 609, **611**, 612, 622, 623, 624
- S**
- SalariedEmployee class**
 header 563
 implementation file 564
Salary Calculator exercise 137
Sales Commission Calculator exercise 137
Sales Summary exercise 302
Salesperson Salary Ranges exercise 299
same_as concept (C++20) **869**
sample algorithm 837
samples (in datasets) 418
savings account 153
SavingsAccount class 512
scanning images 5
schedule a task to execute 1084
scheduling threads 1001
science, technology, engineering and math (STEM) xxiv
scientific computing 21
scientific notation **117**, 1105, 1120, 1121, **1129**
scientific stream manipulator 1111, **1120**
scope **239**, 945
 class **239**
 example 241
 file **240**
 function **239**
 function parameter **239**
 namespace **239**
 of an identifier 1182
scope of a variable **149**
scope resolution operator (::) 211, 213, 451, 488, 854, 946, 1172, 1181
scoped enumeration (enum class) **210**
scoped_lock class **1023**
scraping 400
screen 4, 5, 27, 67
screen-manager program 545
scrutinize data 451
SDK (Software Development Kit) **48**
search algorithms 1198
 binary search **1203**
 recursive binary search 1223
 recursive linear search 1223
 search algorithm 837
 search_n algorithm 837
search key **1201**, 1203
search pattern (regular expressions) 399
searching 699, 796, 1198
 arrays xxx, **277**
second data member of pair **733**
second refinement in top-down, stepwise refinement 120
secondary storage 7
 unit **6**
secondary storage device **358**
second-degree polynomial 78
"secret" implementation details 1184
secret-key cipher 494
secret-key cryptography xliv

- secure password validation 412
- security xxix, xl, 64, 129, 246
 - best practices xxvii
 - flaws 264
- seed
 - the random-number generator 207, 208
- seek
 - direction 383
 - get pointer 383
 - put pointer 383
- seekg function of `istream` 383
- seekp function of `ostream` 383
- selection sort 1213, 1219
 - algorithm 1211
 - efficiency 1210
 - with call-by-reference 1211
- selection statement xxix, 97, 98, 99, 112, 177, 179
 - `if` 98, 100, 179
 - `if...else` 98, 101, 102, 116, 179
 - `switch` 98, 164, 179
 - with initializer 165
- self-assignment 624
 - in operator overloading 594
- self-documenting 73
- self-driving car 655
- self-driving cars 53, 54
- semaphore 1058
 - acquire 1059
 - release 1059
- <semaphore> header (C++20) 203, 1059
- semicolon (;) 68, 82
- send a message to an object 24
- sentiment analysis 53, 425
- sentinel (C++20 ranges) 722
- sentinel-controlled iteration xxix, 113, 115, 116, 185
- sentinel value 112, 113, 116
- separate interface from implementation 448
- Separating Digits exercise 251
- separating interface from its implementation 1161
- `seq` execution policy 995
- sequence 97, 99, 176, 179, 278, 710
 - sequence container 704, 704, 712, 715, 721, 725, 742
 - `back` function 721
 - `empty` function 722
 - `front` function 721
 - `insert` function 721
 - sequence of random numbers 207
- `sequenced_policy` class 995
- sequential file 378, 379, 381, 382, 385
- serialization
 - avoid language native serialization 516
 - pure data formats 516
 - security 516
- Serialization with JSON exercise 515
- serializing data 516
- `set` associative container 705, 730, 734
- `set` function
 - validate data 442
- <`set`> header 201, 730, 734
- `set_new_handler` function 676, 679
- `set` operations of the standard library 807
- `set_difference` algorithm 837
 - ranges version (C++20) 807, 808
- `set_intersection` algorithm 837
 - ranges version (C++20) 807, 808
- `set_symmetric_difference` algorithm 837
 - ranges version (C++20) 807, 809
- `set_union` algorithm 837
 - ranges version (C++20) 809
- `setbase` stream manipulator 1112
- `setfill` stream manipulator 1117, 1118
- `setprecision` stream manipulator 117, 1112
- `setw` stream manipulator 1114, 1117
- SFINAE (substitution failure is not an error) 878
- shadow a data member 482
- Shakespeare xxxii, 426, 427
- shallow copy 619
- Shape class hierarchy 534, 583
- share data 989
- shared buffer 1009
- shared mutable data 989, 1016
- `shared_lock` class 1036
- `shared_mutex` class 1036
- <`shared_mutex`> header 202
- `shared_ptr` class 601, 1143, 1143, 1144, 1146, 1147, 1151, 1153
 - custom deleter function 1147
 - memory leak in circularly referential data 1152
 - `reset` member function of class `shared_ptr` 1147
 - `use_count` member function of class `shared_ptr` 1146
- `shared_ptr` smart pointer
 - custom deleter 1147
- shell prompt on Linux 3, 28
- `shift_left` algorithm 837
- `shift_right` algorithm 837

- shifted, scaled integers 205
- “shipping” section of the computer 5
- shopping list 106
- short-circuit evaluation **170**, 862
- `showbase` stream manipulator 1111, **1119**
- `showpoint` stream manipulator **1116**
- `showpos` stream manipulator 1111, **1118**
- `shrink_to_fit` container member function for `vector` and `deque` **719**
- `shrink_to_fit` member function of classes `vector` and `deque` **719**
- `shuffle` algorithm 837
 - ranges version (C++20) 792, **793**
- side effect 219
 - of an expression 219, **240**
- sides of a right triangle 141
 - exercise 141
- sides of a triangle 141
 - exercise 141
- Sieve of Eratosthenes 751
 - exercise 307
 - exercise with `bitset` 751
- signal
 - a latch 1052
 - operation on semaphore 1059
- signal value **112**
- signature **198**, 225
 - of overloaded prefix and postfix increment operators 632
 - overriding a base-class `virtual` function 552
- significant digits 1116
- silicon 4
- SIMD (single instruction, multiple data) instructions 991
- similarity detection xxxi, 53, **425**, 426
- Simple compiler case study xxxiv, 759
 - 761**
 - data counter **765**
 - enhancements 770
 - first pass 759
 - goal **761**
 - made up programming language 753
 - optimize 769
 - production **769**
 - second pass 759
 - Simple programming language 756
 - symbol table **759**
 - token **760**
 - unresolved forward reference 761
- simple condition 169
- Simple Decryption exercise 410
- Simple Encryption exercise 410
- simple linear regression xxxi, 420
- simple requirement in C++20 concepts 874, **874**
- Simple Sentiment Analysis exercise 415
- `simple_ordinal_nary_least_squares` function from the Boost Math library **421**
- simplest activity diagram 176, 177
- Simpletron computer case study xxx
 - simulator 348, 351, 354
- Simpletron Machine Language xxx, xxxiv
- Simpletron Machine Language (SML) **349**
- Simpletron simulator
 - modifications 354
- Simpletron virtual machine 753, 756
- simulation xl, xliv
 - techniques xxix
- Simulation: Tortoise and the Hare exercise 347
- `sin` function 192
- sine 192
- single-argument constructor 642, 643
- single entry point 176
- single-entry/single-exit control statements **99**, 176
- single exit point 176
- single inheritance **533**, 533, 1173, 1174
- single instruction, multiple data (SIMD) instruction 991
- single-line comment **67**
- single-precision floating-point number **156**
- single quote (‘) 69, 340
- single-selection statement **98**, 100, 179
 - `if` 100
- single-threaded application 989
- single-use gateway 1052
- singly linked list 705, **705**, 724
- six-sided die 204
- size
 - of a `string` 365
 - of a `vector` **716**
 - of an `array` 329
- size global function 291
- size member function
 - `array` **261**
 - containers 709
 - `initializer_list` 618
 - `priority_queue` 743
 - `queue` 742
 - `stack` 740, 852
 - `string` **365**
 - `string_view` **377**
 - `vector` **291**

- size of a `string` **365**
 size of a variable **75**, 1182
`size_t` type **263**, 329
`size_type` **707**
`sizeof` operator **329**, 329, 411
 applied to an array name
 returns the number of bytes in the array 329
 exercise 411
 used to determine standard data type sizes 330
`sizeof...` operator **894**
 skipping whitespace 1111
`skipws` stream manipulator 1111
`sleep` interval **1000**
`sleep_for` function of the `std::this_thread` namespace **1005**
`sleep_until` function of the `std::this_thread` namespace **1005**
 sleeping thread **1000**
 slicing (logic error) **661**
 slope **420**
 small circles in the UML **97**
`smallest` **196**
 smallest of several integers 184
 smart contracts 46, 47
 smart homes 53
 smart pointer **601**, 1143
 `shared_ptr` class **1143**, 1143, 1144, 1146, 1147, 1151, 1153
 `unique_ptr` **605**, 1143
 `weak_ptr` class 1147
 smart pointers
 `make_unique` function template **601**, 603
 smart thermostats 53
 smart traffic control 53
 smartmeters 53
 smartphone 17
`smatch` **405**
`str` member function **407**
- smiley face emoji 410
 SML (Simpletron Machine Language) **349**, 351, 354
 instruction **349**
 software xxiv, xxviii, **2**
 software-based simulation xxx, 348, **351**
 Software Development Kit (SDK) **48**
 software engineering
 case studies xxxv
 information hiding **437**
 observations xxvii
 reuse 85, 434, 447
 separate interface from implementation **448**
 software model **351**
 software reuse **20**, 191, 849
 solid circle in the UML **98**
 solid circle surrounded by a hollow circle in the UML **98**
 solid-state drive 4
 solution 29
Solution Explorer 29, 30
 solve problems xl
`sort` algorithm **277**, 278, 636, 837, 991
 ranges version (C++20) 796, **797**, 826, 827, 828
 sort algorithms
 bubble sort 1221, 1222
 bucket sort 1222
 insertion sort **1209**, 1210
 merge sort **1213**
 quicksort 1223
 selection sort **1211**
 sort key 1198
`sort` member function of `list` **725**
`sort` standard library function 1207
`sort_heap` algorithm 837
 ranges version (C++20) **818**
 sorting 699, 796, 1198
 arrays xxx
 arrays **277**
 order 797, 803
 related algorithms 836
 strings 202
 sorting algorithms 1208
 source code **14**, **25**
 space-time trade-off **718**
 spaces for padding 1118
 spaceship operator (`<=>`) xxxiii, 636
 spam detection 53
 Spam Scanner exercise 413
`span` class template (C++20) 311, **334**
 back member function **338**
 first member function **338**
 front member function **338**
 last member function **338**
`subspan` member function **338**
`` header (C++20) 203, 311, **334**
 spdlog logging library 685
 speaking to a computer 5
 special characters 72
 special member functions xxxiii, **441**, 589, 605, **605**
 constructor 438, 441
 containers 707
 copy assignment operator xxxiii, 441, 589, **592**, 621
 copy constructor xxxiii, 441, 589, 620
 destructor xxxiii, 441, 463, 589
 move assignment operator xxxiii, 441, 589
 move constructor xxxiii, 441, 589
 remove with `= delete` 619
 Rule of Five 619
 Rule of Zero **619**

- special symbol **10**
 specialized memory algorithms **838**
 specialized memory operations **836**
 speech recognition **425**
 speech synthesis **425**
 spell checking **425**
 spelling correction **425**
 spiral **234**
splice member function of *list* **725**
splice_after member function of class template *forward_list* **725**
split range adaptor (C++20) **829**
 split the array in merge sort **1213**
 spooling **1008**
 spurious wakeup **1021**
sqrt function of *<cmath>* header **192**
 square **140**
Square Class (exercise) **508**
square function **199**
 Square of Asterisks exercise **140**
 square root **192, 1113**
<sstream> header **202, 386, 386**
stable_partition algorithm **838**
stable_sort algorithm **838**
stack **214**, **699**
 frame **238**
 overflow **233**
stack adapter class **706, 740**
emplace function **740**
empty function **740**
pop function **740**
push function **740**
size function **740**
top function **740**
Stack class template **849, 852, 853**
 stack data structure **233, 849**
 stack frame **214**
 stack frames **233**
<stack> header **201, 740**
 stack overflow **215**
 stack unwinding **662**, **667, 676, 696**
 stackful coroutine **1080**
 stacking
 building blocks **179**
 control statements **179**
 rule **177**
 stackless coroutine **1080**
StackOverflow (*stackoverflow.com*) **xxxviii, xlvi**
 stale value **1016**
 standard C++20 concepts by header **862**
 standard concepts (C++20) **860**
 standard data type sizes **330**
 standard deviation **418**
 standard document (C++) **xlvi**
 standard error stream (*cerr*) **28**
 standard exception classes **681**
bad_alloc **676**, **681**
bad_cast **681**
bad_typeid **681**
exception **681**, **681**
invalid_argument **682**
length_error **682**
logic_error **681**
out_of_range **682, 682**
overflow_error **681**
runtime_error **658, 668, 681**
underflow_error **681**
 standard input stream (*cin*) **27, 1103**
 standard input stream object (*cin*) **378**
 standard library **191**
 class **string** **84, 591**
deque class template **729**
 exception hierarchy **680**
 standard library (cont.)
 headers **202**
list class template **724**
map class template **738**
multimap class template **736**
multiset class template **731**
priority_queue adapter class **744**
queue adapter class templates **742**
set class template **735**
stack adapter class **740**
vector class template **716**
 standard library algorithms
 minimum requirements **774**
 standard library exception
filesystem_error **684**
 standard library exception hierarchy **680**
 standard output object (*cout*) **68, 1103**
 standard output stream (*cout*) **27**
 standard output stream object (*cout*) **378**
 standard stream libraries **1103**
 Standard Template Library (STL) **698**
Start Window in Visual Studio Community Edition **29**
starts_with member function of class **string** (C++20) **86**
starts_with member function of **string_view** **377**
 starvation **1001**
 state bits **1124**, **1126**
 state dependent **1009**
 statement **68, 113, 756**
break **163**, **167, 187**
continue **167**, **187**
 control statement **95, 97, 99**

statement **68**, 113, 756
 control-statement nesting **99**
 control-statement stacking **99**
 do...while 98, **158**, 179
 double selection **98**, 120
 empty 104
 for 98, **148**, 156, 179
 if **79**, 98, 100, 179
 if...else 98, **101**, 102, 116, 179
 iteration **97**, 105
 looping **98**
 multiple selection **98**
 nested control statements **118**
 nested if...else **102**, 139
 repetition **98**
 return 69, 70
 selection **97**, 98
 single selection **98**
 spread over several lines 82
 switch 98, 159, 164, 179
 throw **451**
 try **295**
 while 98, 106, 109, 116, 147, 179
static
 data member **487**, 488
 data member tracking the number of objects of a class 490
 keyword **213**, 240, 1183
 local object 464, 465, 467
 local variable 240, 243, 782
 local variable thread-safe initialization 1047, 1048
 member 488
 member function **488**
static binding **553**
static code analysis tools xxxvii
 clang-tidy xxxvii, lvii
 cppcheck xxxvii, xxxviii, lvii
 lvii

static polymorphism (compile-time) **847**, 849
static storage class **1183**
static storage duration 1182, 1183
static storage-class specifier **1182**
static_assert declaration **879**
static_cast operator **644**
statistical thinking xl
statistics
 count **418**, 751
 maximum **418**, 751
 measures of central tendency **396**, **418**
 measures of dispersion **418**
 measures of variability **418**
 minimum **418**, 751
 range **418**, 751
 standard deviation 418
 sum **418**, 751
 variance 418
std namespace **69**
std::add_const metaprogram **923**
std::advance function **872**
std::as_const function **896**
std::async function template 1040, **1046**
std::atomic
 load function **1051**
std::atomic class template **1049**
std::atomic type 1049
std::atomic_ref class template (C++20) **1049**, 1049, 1052
std::barrier (C++20) 1052, **1055**
std::begin function 323
std::binary_semaphore (C++20) **1059**
 acquire member function **1061**, 1062
 release member function **1061**, 1062
std::call_once **1048**
std::chrono namespace 993
std::chrono::duration class **993**
std::cin 74
std::condition_variable class **1019**
 notify_one function **1021**, 1022, 1023, 1031
std::condition_variable any class **1037**
 notify_all function **1037**
std::counting_semaphore (C++20) **1059**
std::cout 68
std::distance function **872**
std::end function 323
std::exclusive_scan parallel algorithm **998**
std::execution::parallel_execution_policy **994**, 994, **995**, 995, **995**, 995
std::execution::par_unseq execution policy **995**
std::execution::parallel_policy class **995**
std::execution::parallel_sequenced_policy class **995**
std::execution::seq execution policy **995**
std::execution::sequenced_policy class **995**
std::execution::unseq execution policy **995**
std::execution::unsequenced_policy class **995**
std::filesystem::path **380**
std::floating_point concept **861**, 868

std::for_each_n parallel algorithm **998**
std::forward_list class template 705
std::future class template 1046
std::hash 730
std::inclusive_scan parallel algorithm **998**
std::initializer_list class template **618**
std::integral concept **861**, 868
std::invoke function 827
std::iota algorithm 838
std::jthread (C++20)
 join function **1007**
 request_stop function **1039**, 1040
std::latch (C++20) 1052, **1052**, 1052, 1053
 count_down member function **1053**
 wait member function **1053**, 1053
std::launch enum
 async 1046
 deferred 1046
std::lock_guard class **1023**
std::make_unique function template **601**, 603
std::mdspan (C++23) 1192
std::minmax algorithm **812**
std::move function **612**, 612
std::mutex class **1019**, 1020
std::numeric_limits::max() 993
std::once_flag **1048**
std::optional class 311
std::optional class template 311
std::packaged_task function template **1047**
std::promise **1047**
std::quoted stream manipulator **384**
std::random_device random-number source **208**, 213
std::ranges namespace 699, 774
 all_of algorithm 796, **798**
 any_of algorithm 796
std::ranges namespace (C++20) **722**, 778, 779, 781, 783, 786, 790, 792, 796, 800, 802, 805, 807, 810, 812, 817
std::ranges::count_if algorithm (C++20) **396**, 397
std::ranges::distance algorithm 883
std::reduce parallel algorithm **998**
std::reverse_iterator iterator adaptor **892**
std::same_as concept (C++20) **869**
std::scoped_lock class **1023**
std::shared_future class template 1047
std::shared_lock class **1036**
std::shared_mutex class **1036**
std::shared_ptr class template 601
std::size global function 291
std::stop_callback (C++20) **1040**
std::stop_source for cooperative cancellation (C++20) **1039**
std::stop_token for cooperative cancellation (C++20) **1039**
 stop_requested function **1039**
std::string_literals **277**
std::string_view 310, **374**, 436
std::this_thread namespace
 yield function 1050
std::this_thread::get_id function **1004**
std::this_thread::sleep_for function **1005**
std::this_thread::sleep_until function **1005**
std::thread **1003**
std::thread class **1003**
std::thread::id **1004**
std::to_string function **373**
std::transform_exclusive_scan parallel algorithm **998**
std::transform_inclusive_scan parallel algorithm **998**
std::transform_reduce parallel algorithm **998**
std::unique_lock class **1020**, **1021**, 1021
std::unique_ptr class template 601, **601**, **603**
std::unordered_multimap class template 706
std::unordered_multiset class template 706
std::unordered_set class template 706
std::variant class template **1154**
 for runtime polymorphism **1154**
std::visit standard library function **1154**, 1159
std::weak_ptr class template 601
std::jthread (C++20) **1003**, 1003, **1008**

std.core in the Microsoft modularized standard library 967

std.filesystem in the Microsoft modularized standard library 967

std.memory in the Microsoft modularized standard library 967

std.regex in the Microsoft modularized standard library 968

std.threading in the Microsoft modularized standard library 968

<stdexcept> header 201, 658, 681
 out_of_range 630

steady_clock class 993

stemming 426

sticky setting 117

sticky stream manipulator 1112

STL (Standard Template Library) 698

stock market forecasting 53

stod function 373

stof function 373

stoi function 373, 765, 769

stol function 373

stold function 373

stoll function 373

stop word elimination 426

stop_requested member function of a
 std::stop_token 1039

<stop_token> header (C++20) 203, 1037, 1052

storage-class specifiers 1182
 extern 1182
 mutable 1182
 static 1182

storage duration 1182, 1182
 automatic 1182
 dynamic 1182
 static 1182
 thread 1182

store 769

stoul function 373

stoull function 373

str member function of an **smatch** 407

str member function of class **ostringstream** 386, 387

straight-line form 77

stream base 1112

stream extraction operator >> 74, 81, 226, 588, 633

stream extraction operator >> ("get from") 1104, 1106

stream input/output 67

stream insertion operator << 69, 70, 74, 226, 381, 588, 633, 1104

stream manipulator 1103, 1111
 boolalpha 86
 custom 1115
 quoted 384
 sticky 1112

stream manipulators 117
 boolalpha 1122
 dec 1112
 fixed 117, 1113, 1120
 hex 1112
 internal 1117
 left 1117
 noboolalpha 1122
 noshowbase 1119
 noshowpoint 1116
 noshowpos 1111, 1118
 noskipws 1111
 nouppercase 1111, 1121
 oct 1112
 right 1117
 scientific 1120
 setbase 1112
 setfill 1118
 setprecision 117, 1112
 setw 1114
 showbase 1119
 showpoint 1116
 showpos 1118
 skipws 1111

stream of bytes 1102

stream of characters 68

stream operation failed 1125

streaming 989

<string> header 201

string 833
 C style 310
 pointer based 310
 processing xxxi

string
 iterators 786

string built-in type
 in JSON 516

string class 83, 85, 201, 436, 589, 591, 705
 assignment 359
 at member function 595
 concatenation 359
 concatenation exercise 409
 empty member function 86, 592
 ends_with member function (C++20) 86
 find functions 368
 find member function 368
 insert functions 372
 insert member function 372
 length member function 85
 starts_with member function (C++20) 86
 subscript operator [] 360
 substr member function 593

string concatenation 86, 905

string concatenation assignment 592

string formatting 152

<string> header 85

string literal 68
 raw string literal 389

string object literal 277, 592, 833, 855

string of characters 68

- string processing 191
string stream processing **386**
string_literals **277**
string_view class 310, **374**, 436
 find member function **377**
 remove_prefix member function **376**
 remove_suffix member function **376**
 size member function **377**
 starts_with member function **377**
<string_view> header 374
string::npos **368**
string-object literal **592**
 strings as full-fledged objects 340
 strong encapsulation **948**, 970
 strong exception guarantee 621
 copy-and-swap idiom 664
 strong exception safety guarantee **664**
 Stroustrup, B. 18
 Stroustrup, Bjarne 19
struct **492**
 structured binding 812, 813
 unpack elements **795**
 structured binding declaration **795**
 structured programming **96**, 146, 168, 176
 summary 175
 Student Inheritance Hierarchy 583
 student-poll-analysis program 276
 subclass **530**
submit function of a **currenccpp** executor **1084**
 subobject of a base class 1174
 subscript **261**
 subscript operator 729
 map 738, 739
 subscripted name used as an *rvalue* 610
 subscripting 728
subspan member function of **span** class template (C++20) **338**
 substitution cipher 246, **246**, 494, 497
substr 363
substr member function of class **string** **363**, **593**
 substring of a **string** 363
 subtract pointers 331
 subtraction 6, 76, 77
 subtraction compound assignment operator, **-=** 125
 sufficient conditions for deadlock 1002
suffix member function of class **match_results** **407**
 sum of the elements of an array 271, 284
 sum statistic **418**, 751
 summarizing text 53
 superclass **530**
 supercomputer **7**
 super-sized integers 129
 survey 275, 276
 suspend a coroutine's execution 1079
suspend_always (coroutines) **1094**
suspend_never (coroutines) **1094**
 suspension point (coroutines) **1095**
 Sutter, Herb 19, 1161
 blog 670, 687
 ISO C++ Convener 687
swap algorithm **800**
swap member function
 of class **string** **364**
 of class **unique_ptr** 636
 of containers 709
 of **list** **727**
 standard library function 636
swap_ranges algorithm 800, 837
 ranges version (C++20) **801**, 801
 swapping strings 364
 swapping values 1208, 1210
 Swift programming language 17, 22
switch logic 165
switch multiple-selection statement 98, **159**, 164, 179
 case label 162, 163
 controlling expression **162**
 default case **162**, 164
switch with initializer 165
 symbol table 760
 Simple compiler **759**
 symmetric encryption **521**
 symmetric key encryption **410**
 synchronization **1016**, 1017
 synchronization point 1052
 synchronize operation of an *istream* and an *ostream* 1127
 synchronized block of code 1019
 synchronized threads **989**
 synchronous error **669**
syntax **68**
 syntax coloring conventions in this book xlivi
 syntax error **68**
system_clock class **993**
 systems programming case studies xliii
 systems programming case study exercises
 Building Your Own Compiler xxx, xxiv
 Building Your Own Computer xxiv
T
Tab key 68
 tab stop 69

- table of values 279
 tablet computer 17
 tabular format 264
 Tabular Output exercise 137
 tag dispatch 878
 tail of a queue 699
 tails 204
take range adaptor (C++20) 829, 831
take_while range adaptor (C++20) 829, 831
tan function 192
 tangent 192
task (concurrency) 1081
 task for asynchronous operations 1076
 TCP (Transmission Control Protocol) 41
 TCP/IP 41
 Telephone Number Word Generator 302, 411
 Telephone Number Word Generator exercise 411
tellg function of **istream** 383
tellp function of **ostream** 383
 template deduction guide 893
 default type argument for a type parameter 898
 type argument 850
 template definition 228
 template function 228
template header 850
 template instantiations 227
template keyword 227, 850
 template metaprogramming (TMP) xxvii, 18, 848, 913
 metafunction 916
 Turing complete 914
 type metafunction 917
 value metafunction 917
template parameter 850
template parameter list 227
 templated lambda (C++20) 856
 templated lambda expression 854
 templates defining in C++20 modules 945
 partial specialization 924
 variable template 898
 temporary value 199
 terabit Internet 41
 terabytes (TB) 6, 50
 teraflop 9
 terminate a loop 113
 terminate a program 679
 terminate normally 380
terminate standard library function 668
 terminate successfully 69
 terminated state 1001
 terminating condition 231
 terminating right brace (}) of a block 239
 termination
 abort function 668, 679
 exit function 679
 terminate function 668
 termination housekeeping 463
 termination model of exception handling 662
 termination test 238
 ternary operator 104
test 745
 test characters 201
 Test-Driven Development (TDD) 49
 Testing Regular Expressions Online exercise 412
 text editor 381
 text formatting (C++20) xxix, 1127, 1128, 1129
 # to display a number's base 1131
 + to display a sign 1131
 0 to fill a field with leading 0s 1131
 a (or A) presentation type 1129, 1130
 text formatting (C++20) (cont.)
 c presentation type 1130
 d (integer) 156
 d presentation type 1130
 e (or E) presentation type 1129
 f (floating-point number) 155
 f presentation type 1129
 field width 155
 fill with 0s 1131
 format specifier 155
 format string 153
 g (or G) presentation type 1129
 left align (<) 1130, 1131
 nested placeholder 1132
 not yet supported by Boost Multiprecision 182
 numbers with their signs (+) 1131
o presentation type 1130
 placeholder 153, 155
 precision of a floating-point number 155
 presentation type 1128, 1129
 right align (>) 1130, 1131
 right align > 155
 space in a format specifier 1131
x (or X) presentation type 1130
 text-printing program 67
 text summarization 425
 the cloud xxiv, 42, 515
 theft prevention 53
 Thinking Like a Developer xxxviii
this pointer 482, 483, 491, 622, 625
 used explicitly 482
 used implicitly and explicitly to access members of an object 483

- thread **989**
 - exception **1003**
 - of execution **989**
 - scheduling **1000**, 1014
 - state **999**
 - synchronization **1016**
- thread class **1003**
- thread-coordination primitives **1048**
- thread-coordination types **1052**
- `<thread>` header **202**, **1003**
- thread launch policy **1046**
- thread-local storage
 - thread safe **990**
- thread pool **1081**
- thread safe **989**, **1015**, **1019**
 - one-time initialization **1047**
 - atomic type **989**
 - immutable data **989**
 - linked data structures **1051**
 - mutual exclusion **989**
 - thread local storage **990**
- thread scheduler **1001**
- thread states
 - blocked **1001**
 - born **1000**
 - ready **1000**
 - running **1000**
 - terminated **1001**
 - timed waiting **1000**
 - waiting **1000**
- thread storage duration **1182**
- thread synchronization
 - coordination types **1052**
- `thread_executor` (concurrency) **1084**
- `thread_local` storage class **990**
- `thread_pool_executor` (concurrency) **1081**, **1084**
- `thread::id` **1004**
- three-way comparison operator (`<=>`; C++20) **xxxiii**, **203**, **441**, **442**, **636**, **637**, **708**
- `throw` **663**, **681**
 - standard exceptions **681**
- `throw` an exception **294**, **295**, **451**, **660**
 - from a constructor **672**
- throw point **661**, **668**
- Throwing Exceptions from a catch **696**
- `TicTacToe` Class exercise **515**
- `tie` an input stream to an output stream **1127**
- tightly coupled **573**
- tilde character (`~`) **463**
- time-and-a-half **137**
- time and date utilities **993**
- `Time` class **512**
 - constructor with default arguments **457**
 - definition **449**
 - definition modified to enable cascaded member-function calls **484**
 - member-function definitions **450**
 - member-function definitions, including a constructor that takes arguments **458**
- `Time` Class Modification **509**
- time series **xxxii**, **423**
 - Climate at a Glance **423**
 - observations **xxxii**, **423**
- timed waiting* thread state **1000**
- timer for performing a task in the future **1076**
- timeslice **1000**, **1001**
- Timing 60,000,000 Die Rolls exercise **1069**
- timing operations **xxxv**
- Titanic* disaster dataset **359**, **392**
- `t1::generator` class template (generator library) **1077**
- TMP (template metaprogramming) **848**, **913**
- `to_array` function of header `<array>` (C++20) **311**, **324**
- `to_string` function **373**
- token **384**, **760**
- tokenization **425**
- tokens **425**
 - Simple compiler **760**
- `top` **112**
- top-down, stepwise refinement **xxix**, **112**, **113**, **114**, **119**, **120**
- `top` member function
 - of `priority_queue` **743**
 - of `stack` **740**
- `top` member function of a stack **851**
- top of a stack **699**
- Tortoise and the Hare exercise **347**
- total **108**, **112**
- totient **523**
- Towers of Hanoi **253**
 - Iterative Version **254**
- trailing `requires` clause **869**
- Trailing `requires` Clause and an Ad-Hoc Constraint exercise **929**
- trailing return types **781**, **1187**
- trailing zeros **1116**
- transaction processing **736**
- transfer of control **349**, **353**
- `transform` algorithm **837**
 - ranges version (C++20) **792**
- `transform` range adaptor (C++20) **829**, **832**
- `transform_exclusive_scan` parallel algorithm **838**, **998**

- t** `transform_inclusive_scan` parallel algorithm 838, 998
`transform_reduce` algorithm 838
`transform_reduce_parallel_algorithm` 998
 transforming data 400
 transition arrow in the UML 97
 transition from the preprocessor to modules 938
 translate speech 54
 translation 13, 26
 translation services
 Bing Microsoft Translator 413
 Google Translate 413
 translation unit 899, 936, 939, 958, 1180
 non-module 951
 part of a module 943
 translator program 13
 Transmission Control Protocol (TCP) 41
 trapdoor function 528
 treat warnings as errors 221
 trend spotting 53
 Triangle-Printing Program
 exercise 185
 modified 186
 trigonometric cosine 192
 trigonometric sine 192
 trigonometric tangent 192
 triple indirection 570
`tripleByReference` 256
`tripleByValue` 256
 trivially copyable type 1051, 1052
`true` 80, 100
`truncate` 76, 379
 fractional part of a calculation 110
 fractional part of a `double` 198
 truth tables for logical operators 169, 170
`try block` 295, 665, 668
 expiration 662
 nested 667
`try statement` 295
`tuple`
 pack 901, 901
 unpacking 901
`tuple class template` 899
 getting a tuple member 901
 `make_tuple` function 901
`<tuple>` header 201, 899
 tuples of Student Grades exercise 930
 Turing complete 914
 Turing test 58
`tvOS` 16
 two-dimensional array 279, 279
 two largest values 138
 type alias 900, 901
 type argument 603, 821, 850
 type category 868
 type dependent formatting 1129
 type metafunction 917
 predefine 924
 type name
 alias 899, 1157
 type of a variable 75, 1182
 type of the `this` pointer 482
 type parameter 227, 228, 850
 type requirement in C++20
 concepts 860, 874, 875
 type-safe linkage 225
 type-safe union 1158
 type trait 848
 `is_base_of` 919
 `value` member 866
`<type_traits>` header 921, 864
`typedef`
 `iostream` 1104
 `istream` 1104
 `ostream` 1104
`typeid` 681
`<typeinfo>` header 201
`typename` keyword 227, 850
`typename...` in a variadic template 903
 types of programming languages xxviii
- U**
- Ubuntu Linux 32
 in the Windows Subsystem for Linux 32
 UML (Unified Modeling Language)
 activity diagram 97, 97, 106, 158
 arrow 97
 diamond 101
 dotted line 98
 final state 98
 guard condition 101
 merge symbol 106
 note 98
 solid circle 98
 solid circle surrounded by a hollow circle 98
 UML class diagram 533
 unary left fold 906, 908
 unary operator 171, 313
 unary operator overload 597
 unary predicate function 789, 791
 unary right fold 906, 908
 unary scope resolution operator `(::)` 244
 unbuffered output 1104
 unbuffered standard error stream 1103
 uncaught exception 667, 668
 uncaught exceptions 696
 unconditional branch 768
 unconstrained function template 857
 undefined behavior 110, 314, 599, 620
 division by zero 658
 undefined value 441

underflow_error exception **681**
underlying data structure 743
underscore (_) 73
unformatted I/O **1103**, 1110
unformatted output 1104,
 1106
unhandled_exception
 function of a coroutine
 promise object **1094**
Unicode character set **10**, 90,
 1103
uniform_int_distribution **204**
unincremented copy of an ob-
 ject 632
union **1158**
unique algorithm 837
 ranges version (C++20)
 802, **804**
unique keys 734, 738
unique member function of
 list **727**
unique_copy algorithm 837
 ranges version (C++20)
 805, **806**
unique_lock class **1020**, 1021
 unlock function **1022**
unique_ptr 605
unique_ptr class 601, **601**,
 604, 605, 1143
 built-in array **603**
 create with **make_unique**
 function template 621
 get member function 618
 swap member function 636
univariate time series **423**
universal-time format 451
University of Tennessee Mar-
 tin Prime Pages website
 1041
UNIX 162, 380
unlock function of a
 unique_lock **1022**
unnamed namespace **1180**
unordered associative con-
 tainers **704**, **706**, **708**, 730
unordered_map associative
 container class template
 706, 730, 738
<unordered_map> header
 201, 736, 738
unordered_multimap associative
 container class template
 706, 730, 736
unordered_multiset associative
 container class template
 706, 730, 731
unordered_set associative
 container class template
 706, 730, 734
<unordered_set> header
 201, 731, 734
unpack elements (C++17
 structured binding) **795**
unpack elements via struc-
 tured binding **795**
unpacking a tuple **901**
unresolved forward reference
 (Simple compiler) 761
Unruh, Erwin 914
unseq execution policy **995**
unsequenced_policy class
 995
unsigned data type 199
 integer types **198**
 unsigned char 199
 unsigned int 199
 unsigned long 199
 unsigned long int 199
 unsigned long long 199
 unsigned long long int
 199
 unsigned short 199
 unsigned short int 199
untie an input stream from an
 output stream 1127
unwinding the function call
 stack 667
update records in place 385
upper_bound algorithm 837
 ranges version (C++20)
 810, **811**

upper_bound function of as-
 sociative container **733**
uppercase letter 72, 90, 201
uppercase stream manipula-
 tor 1111, 1119, **1121**
use cases 53
use_count member function
 of class **shared_ptr** **1146**
user defined
 function 194
 type 210, 211
user-defined type **434**, 640
using a dynamically allocated
 ostringstream object
 386
using a function template 227
using a **static** data member
 to maintain a count of the
 number of objects of a class
 488
using arrays instead of
 switch 274
using declaration **81**, 132,
 946
 in headers 436
using declaration to create an
 alias for a type **899**, **900**,
 1157
using directive **81**, 946
 in headers 436
using enum statement 213
using function **swap** to swap
 two **strings** 364
using standard library func-
 tions to perform a heapsort
 817
using virtual base classes
 1176
Utilities area (Xcode) 38
utility function **457**
<utility> header 202
 exchange function **623**

V
V operation on semaphore
 1059
V's of big data 52

validate a first name 402
 validate data 111
 validate data in a *set* function 442
 validating data (regular expressions) 399
 Validating User Input exercise 138
 value initialization **266**, 321, 441, 493, 599, 902
 memory 617
 objects 599
 rules 599
 value member of a type-trait class 866
 value metafunction 917
 value of a variable 75, 1182
 value of an *array* element **261**
 value_type 706, 707
 nested type in an iterator 887
 values range adaptor (C++20) 829, **833**
 variable **72**
 variable name 75, 756, 759
 variable scope 149
 variable size 75
 variable template 866, **898**
 variable type 75
 variadic function template 906
 compile-time recursion 902
 typename... **903**
 variadic template 848, 894, **899**
 parameter pack **894**, 903
 sizeof... operator **894**
 variadic template parameter pack 905
 variance 418
 <variant> header 1154
 variant standard library
 class template 1154
 variety (in big data) 53
 vector 851
 capacity 715
 vector class 290, 670
 vector class template **260**, 704, 716
 capacity function **716**
 crbegin function **719**
 crend function **719**
 push_back function 717
 push_back member function **295**
 push_front function 717
 rbegin function **719**
 rend function **719**
 shrink_to_fit member function **719**
 vector class template element-manipulation functions 720
 vector container
 erase member function 787
 vector hardware operations 991
 <vector> header 201, 290
 vector mathematics **835**, 991
 vectorized execution 996
 velocity (in big data) 52
 veracity (in big data) 53
 Version Control Systems (VCS) **48**
 Git 48
 version control tools xxxix
 video streaming 1027
 view (C++20) **286**, 699, 828
 all range adaptor 829
 common range adaptor 829
 composable **286**
 composing 828
 counted range adaptor 829
 drop range adaptor 829, **832**
 drop_while range adaptor 829, **832**
 elements range adaptor 829, 834
 filter range adaptor 829
 iota range factory 830
 iota range factory for an infinite range **830**
 view (C++20) (cont.)
 keys range adaptor 829, **833**
 reverse range adaptor 829, **831**
 split range adaptor 829
 take range adaptor 829, **831**
 take_while range adaptor 829, **831**
 transform range adaptor 829, 832
 values range adaptor 829, **833**
 view into a container 334
 viewable_range (C++20) **829**
 views of contiguous container elements 334
 views::filter **288**
 views::iota **287**
 Vigenère cipher 502, 505
 Vigenère square **497**
 Vigenère secret key cipher **245**, 246, 248
 Vigenère secret-key cipher xxix, 494
 Vignère secret-key cipher 497
 vim editor 25
 violation handler (contracts) **694**
 default **690**
 virtual base class 1174, 1176
 virtual destructor **557**
 virtual function 531, **552**, 552, 570, 572, 1174
 as an internal implementation detail 1167, 1194
 call 572
 call illustrated 571
 overhead 570
 private 1162
 protected 1162
 table (*vtable*) **570**
 "under the hood" 570

- Virtual Functions exercise
 582
 Virtual Functions vs. Pure
 Virtual Functions exercise
 582
virtual inheritance 1176
 virtual machine xxx, xliv,
 348, 753
 virtual memory 677, 679
 virtual reality (VR) 44, 46
 fully-immersive 45
 non-immersive 44
 semi-immersive 44
 Virtuality (paper by Herb
 Sutter) 1161
 virtualization 49
 visibility (C++20 modules)
 971
visit standard library func-
 tion 1154, 1159
 Visual C++ xxxvii
 Visual C++ compiler xxviii,
 liv, lv
 Visual C++ programming
 language 21
 visual product search 53
 Visual Studio
 Community edition 3
 Visual Studio Code 25
 Visual Studio Code editor 28
 Visual Studio Community
 Edition liv, 26, 28
 Command Prompt win-
 dow 30
**Configure your new proj-
 ect** 29
Create a New Project 29
Empty Project 29
Solution Explorer 29, 30
Start Window 29
 visualization xl, 419
 binary search xxxvi
 merge sort xxxvi
 Visualizing 231
 visualizing recursion 231,
 238, 254
- Visualizing Searching and
 Sorting Algorithms case
 studies xxxvi
 voice recognition 53
void * 333
void return type 195, 196,
 198
 volatile information 6
 volume (in big data) 52
 volume of a cube 218
 VR (virtual reality) 44
vtable 570, 573
 pointer 573
- W**
- W3C (World Wide Web
 Consortium) 42
wait-for (necessary condition
 for deadlock) 1002
wait function of a condi-
 tion_variable 1021
wait member function of a
 std::latch 1053, 1053
 wait operation on semaphore
 1059
 waiting thread 1022
 state 1000
 “walk off” either end of an ar-
 ray 604
-Wall GNU g++ compiler
 flag 174
 “warehouse” section of the
 computer 6
warnings
 treat as errors 221
 watchOS 16
wchar_t character type 1103
weak_ptr class 601, 1147
 bad_weak_ptr exception
 1148
 lock member function of
 class weak_ptr 1148,
 1151
weakly_incrementable
 concept (C++20) 779
 weather forecasting 53
 web scraping 400
- web service 42, 49, 515
 web services
 Any API 44
 APILayer 44
 GitHub Public APIs 43
 Google APIs Explorer 43
 Rapid API 44
 Web3 47
Welcome to Xcode window
 37
 What Does This Code Do?
 exercise 184, 187
 What Does This Program
 Do? exercise 138
 what virtual function of class
 exception 295, 661, 663,
 677
 What’s Wrong with This
 Code? exercise 184
 when to use exceptions xxxiii
when_all function (concur-
 rencpp library) 1088
when_any function (concur-
 rencpp library) 1089
while iteration statement 98,
 106, 109, 116, 147, 179
 whitespace characters 68, 68,
 1106, 1108, 1111
 whole number 72
width member function of
 class ios_base 1114
 width setting 1114
 implicitly set to 0 1114
 William Gates xxxviii
 Williams, Anthony xliv
 Windows 162
 Windows operating system
 15
 Windows Subsystem for Li-
 nux (WSL) 32
wiostream 1104
wistream 1104
 word character 401
 Word Endings exercise 408
 word frequency counting 426
 Word Frequency Counting
 exercise 752

- words **349**
- `worker_thread_executor` (concurrencypp) **1085**
- workflow **97**
- Working with the `dia-monds.csv` Dataset exercise 414
- Working with the `iris.csv` Dataset exercise 415
- workspace window 38
- World Wide Web **42**, 42
- worst-case run time for an algorithm 1199
- `wostream` 1104
- Wozniak, Steve 16
- `write` 761
- `write` function of `ostream` 1105, 1110
- X**
- `X` (or `x`) presentation type **1130**
- `-x c++-module` compiler flag (clang++) **948**
- `-x c++-system-header` compiler flag (g++) **940**
- x86-64 gcc (contracts) 689
- `-xc++-system-header` compiler flag (clang++) **940**
- Xcode xxxvii, liv, 26
- Debug** area 38
 - Editor** area 38
 - Navigator** area 38
 - Utilities** area 38
 - Welcome to Xcode** window 37
- Xcode navigators
- Issue 38**
 - Project 38**
- Xcode on Mac OS X 28
- Xerox PARC (Palo Alto Research Center) 16
- XML (eXtensible Markup Language) 516
- `xor` operator keyword **1185**
- `xor_eq` operator keyword **1185**
- `xvalue` (expiring value) **611**
- y**
- `y-intercept` **420**, 421
- `yield` function of namespace `std::this_thread` 1050
- `yield` the processor 1050
- `yield_value` function of a coroutine promise object **1095**
- Yoda condition 174
- Z**
- zero-based counting 149
- zero-overhead principle of C++ features 656
- zettabytes (ZB) 50

This page intentionally left blank