



DEITEL® DEVELOPER SERIES

Modern C++

Functional-Style Programming

Concepts

Templates

Copy/Move Semantics

Spaceship Operator

Smart Pointers

Text Formatting

Modules

Standard Library

Security

C++20

for
Programmers

An Objects-Natural Approach

Executors

Design Patterns

Coroutines

Ranges/Views

Performance

Contracts

Open Source Libraries

jthread

Parallel Algorithms

Lambdas

Concurrency

PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER



The logo features the letters 'C', '++', and '20' in a bold, 3D-style font. The 'C' is blue, the '++' is blue, and the '20' is red. Below the logo, the text 'for Programmers' is written in a black, sans-serif font.

C++20
for
Programmers

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

O'Reilly Online Learning

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

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

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

C++20 LiveLessons Fundamentals Video Product

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

E-Books

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

Full-Throttle Live Training Courses

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

College Textbook Versions of C++20 for Programmers

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

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



DEITEL® DEVELOPER SERIES

C++20 for Programmers

An Objects-Natural Approach

Paul Deitel • Harvey Deitel



Pearson

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

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

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

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

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

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

Visit us on the Web: <https://informit.com>

Library of Congress Control Number: 2021943762

Copyright © 2022 Pearson Education, Inc.

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

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

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

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

ISBN-10: 0-13-690569-2

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

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

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

Our ambition is to purposefully contribute to a world where:

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

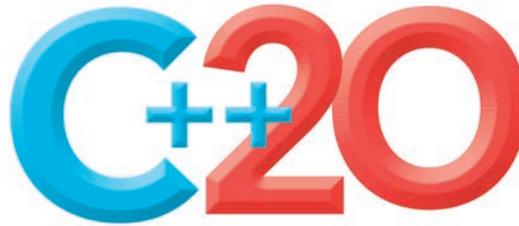
While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them. Please contact us with concerns about any potential bias at

<https://www.pearson.com/report-bias.html>

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

*Paul Deitel
Harvey Deitel*

Contents



Preface **xxi**

Before You Begin **xliii**

1 Intro and Test-Driving Popular, Free C++ Compilers **1**

1.1	Introduction	2
1.2	Test-Driving a C++20 Application	4
1.2.1	Compiling and Running a C++20 Application with Visual Studio 2022 Community Edition on Windows	4
1.2.2	Compiling and Running a C++20 Application with Xcode on macOS	8
1.2.3	Compiling and Running a C++20 Application with GNU C++ on Linux	11
1.2.4	Compiling and Running a C++20 Application with g++ in the GCC Docker Container	13
1.2.5	Compiling and Running a C++20 Application with clang++ in a Docker Container	14
1.3	Moore's Law, Multi-Core Processors and Concurrent Programming	16
1.4	A Brief Refresher on Object Orientation	17
1.5	Wrap-Up	20

2 Intro to C++20 Programming **21**

2.1	Introduction	22
2.2	First Program in C++: Displaying a Line of Text	22
2.3	Modifying Our First C++ Program	25
2.4	Another C++ Program: Adding Integers	26
2.5	Arithmetic	30
2.6	Decision Making: Equality and Relational Operators	31
2.7	Objects Natural: Creating and Using Objects of Standard-Library Class string	35
2.8	Wrap-Up	38

3	Control Statements: Part 1	39
3.1	Introduction	40
3.2	Control Structures	40
	3.2.1 Sequence Structure	41
	3.2.2 Selection Statements	42
	3.2.3 Iteration Statements	42
	3.2.4 Summary of Control Statements	43
3.3	if Single-Selection Statement	43
3.4	if...else Double-Selection Statement	44
	3.4.1 Nested if...else Statements	45
	3.4.2 Blocks	46
	3.4.3 Conditional Operator (?:)	47
3.5	while Iteration Statement	47
3.6	Counter-Controlled Iteration	48
	3.6.1 Implementing Counter-Controlled Iteration	48
	3.6.2 Integer Division and Truncation	50
3.7	Sentinel-Controlled Iteration	50
	3.7.1 Implementing Sentinel-Controlled Iteration	50
	3.7.2 Converting Between Fundamental Types Explicitly and Implicitly	52
	3.7.3 Formatting Floating-Point Numbers	53
3.8	Nested Control Statements	54
	3.8.1 Problem Statement	54
	3.8.2 Implementing the Program	54
	3.8.3 Preventing Narrowing Conversions with Braced Initialization	56
3.9	Compound Assignment Operators	57
3.10	Increment and Decrement Operators	58
3.11	Fundamental Types Are Not Portable	60
3.12	Objects-Natural Case Study: Arbitrary-Sized Integers	61
3.13	C++20: Text Formatting with Function format	65
3.14	Wrap-Up	67
4	Control Statements: Part 2	69
4.1	Introduction	70
4.2	Essentials of Counter-Controlled Iteration	70
4.3	for Iteration Statement	71
4.4	Examples Using the for Statement	74
4.5	Application: Summing Even Integers	74
4.6	Application: Compound-Interest Calculations	75
4.7	do...while Iteration Statement	78
4.8	switch Multiple-Selection Statement	80
4.9	C++17 Selection Statements with Initializers	85
4.10	break and continue Statements	86
4.11	Logical Operators	88
	4.11.1 Logical AND (&&) Operator	88

4.11.2	Logical OR () Operator	89
4.11.3	Short-Circuit Evaluation	89
4.11.4	Logical Negation (!) Operator	90
4.11.5	Example: Producing Logical-Operator Truth Tables	90
4.12	Confusing the Equality (==) and Assignment (=) Operators	92
4.13	Objects-Natural Case Study: Using the <code>miniz-cpp</code> Library to Write and Read ZIP files	94
4.14	C++20 Text Formatting with Field Widths and Precisions	98
4.15	Wrap-Up	100

5 Functions and an Intro to Function Templates 101

5.1	Introduction	102
5.2	C++ Program Components	103
5.3	Math Library Functions	103
5.4	Function Definitions and Function Prototypes	105
5.5	Order of Evaluation of a Function's Arguments	108
5.6	Function-Prototype and Argument-Coercion Notes	108
5.6.1	Function Signatures and Function Prototypes	108
5.6.2	Argument Coercion	109
5.6.3	Argument-Promotion Rules and Implicit Conversions	109
5.7	C++ Standard Library Headers	111
5.8	Case Study: Random-Number Generation	113
5.8.1	Rolling a Six-Sided Die	114
5.8.2	Rolling a Six-Sided Die 60,000,000 Times	115
5.8.3	Seeding the Random-Number Generator	117
5.8.4	Seeding the Random-Number Generator with <code>random_device</code>	118
5.9	Case Study: Game of Chance; Introducing Scoped <code>enums</code>	119
5.10	Scope Rules	124
5.11	Inline Functions	128
5.12	References and Reference Parameters	129
5.13	Default Arguments	132
5.14	Unary Scope Resolution Operator	133
5.15	Function Overloading	134
5.16	Function Templates	137
5.17	Recursion	139
5.18	Example Using Recursion: Fibonacci Series	142
5.19	Recursion vs. Iteration	145
5.20	<code>Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz</code>	147
5.21	Wrap-Up	150

6 arrays, vectors, Ranges and Functional-Style Programming 153

6.1	Introduction	154
6.2	arrays	155
6.3	Declaring arrays	155

✘ Contents

6.4	Initializing array Elements in a Loop	155
6.5	Initializing an array with an Initializer List	158
6.6	C++11 Range-Based for and C++20 Range-Based for with Initializer	159
6.7	Calculating array Element Values and an Intro to <code>constexpr</code>	161
6.8	Totaling array Elements	163
6.9	Using a Primitive Bar Chart to Display array Data Graphically	164
6.10	Using array Elements as Counters	165
6.11	Using arrays to Summarize Survey Results	166
6.12	Sorting and Searching arrays	168
6.13	Multidimensional arrays	170
6.14	Intro to Functional-Style Programming	174
	6.14.1 What vs. How	174
	6.14.2 Passing Functions as Arguments to Other Functions: Introducing Lambda Expressions	175
	6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library	177
6.15	Objects-Natural Case Study: C++ Standard Library Class Template <code>vector</code>	180
6.16	Wrap-Up	187
7	(Downplaying) Pointers in Modern C++	189
7.1	Introduction	190
7.2	Pointer Variable Declarations and Initialization	192
	7.2.1 Declaring Pointers	192
	7.2.2 Initializing Pointers	192
	7.2.3 Null Pointers Before C++11	192
7.3	Pointer Operators	192
	7.3.1 Address (&) Operator	193
	7.3.2 Indirection (*) Operator	193
	7.3.3 Using the Address (&) and Indirection (*) Operators	194
7.4	Pass-by-Reference with Pointers	195
7.5	Built-In Arrays	199
	7.5.1 Declaring and Accessing a Built-In Array	199
	7.5.2 Initializing Built-In Arrays	199
	7.5.3 Passing Built-In Arrays to Functions	199
	7.5.4 Declaring Built-In Array Parameters	200
	7.5.5 C++11 Standard Library Functions <code>begin</code> and <code>end</code>	200
	7.5.6 Built-In Array Limitations	200
7.6	Using C++20 <code>to_array</code> to Convert a Built-In Array to a <code>std::array</code>	201
7.7	Using <code>const</code> with Pointers and the Data Pointed To	202
	7.7.1 Using a Nonconstant Pointer to Nonconstant Data	203
	7.7.2 Using a Nonconstant Pointer to Constant Data	203
	7.7.3 Using a Constant Pointer to Nonconstant Data	204
	7.7.4 Using a Constant Pointer to Constant Data	204
7.8	<code>sizeof</code> Operator	205
7.9	Pointer Expressions and Pointer Arithmetic	208
	7.9.1 Adding Integers to and Subtracting Integers from Pointers	209
	7.9.2 Subtracting One Pointer from Another	209

7.9.3	Pointer Assignment	210
7.9.4	Cannot Dereference a <code>void*</code>	210
7.9.5	Comparing Pointers	210
7.10	Objects-Natural Case Study: C++20 <code>spans</code> —Views of Contiguous Container Elements	210
7.11	A Brief Intro to Pointer-Based Strings	216
7.11.1	Command-Line Arguments	217
7.11.2	Revisiting C++20's <code>to_array</code> Function	218
7.12	Looking Ahead to Other Pointer Topics	220
7.13	Wrap-Up	220

8 strings, `string_views`, Text Files, CSV Files and Regex **221**

8.1	Introduction	222
8.2	<code>string</code> Assignment and Concatenation	223
8.3	Comparing <code>strings</code>	225
8.4	Substrings	226
8.5	Swapping <code>strings</code>	227
8.6	<code>string</code> Characteristics	227
8.7	Finding Substrings and Characters in a <code>string</code>	230
8.8	Replacing and Erasing Characters in a <code>string</code>	232
8.9	Inserting Characters into a <code>string</code>	234
8.10	C++11 Numeric Conversions	235
8.11	C++17 <code>string_view</code>	236
8.12	Files and Streams	239
8.13	Creating a Sequential File	240
8.14	Reading Data from a Sequential File	243
8.15	C++14 Reading and Writing Quoted Text	245
8.16	Updating Sequential Files	246
8.17	String Stream Processing	247
8.18	Raw String Literals	249
8.19	Objects-Natural Case Study: Reading and Analyzing a CSV File Containing <i>Titanic</i> Disaster Data	250
8.19.1	Using <code>rapidcsv</code> to Read the Contents of a CSV File	251
8.19.2	Reading and Analyzing the <i>Titanic</i> Disaster Dataset	253
8.20	Objects-Natural Case Study: Intro to Regular Expressions	259
8.20.1	Matching Complete Strings to Patterns	261
8.20.2	Replacing Substrings	265
8.20.3	Searching for Matches	265
8.21	Wrap-Up	267

9 Custom Classes **269**

9.1	Introduction	270
9.2	Test-Driving an Account Object	271

9.3	Account Class with a Data Member and <i>Set</i> and <i>Get</i> Member Functions	272
9.3.1	Class Definition	272
9.3.2	Access Specifiers <code>private</code> and <code>public</code>	274
9.4	Account Class: Custom Constructors	275
9.5	Software Engineering with <i>Set</i> and <i>Get</i> Member Functions	279
9.6	Account Class with a Balance	280
9.7	Time Class Case Study: Separating Interface from Implementation	283
9.7.1	Interface of a Class	284
9.7.2	Separating the Interface from the Implementation	284
9.7.3	Class Definition	285
9.7.4	Member Functions	286
9.7.5	Including the Class Header in the Source-Code File	287
9.7.6	Scope Resolution Operator (<code>::</code>)	287
9.7.7	Member Function <code>setTime</code> and Throwing Exceptions	287
9.7.8	Member Functions <code>to24HourString</code> and <code>to12HourString</code>	288
9.7.9	Implicitly Inlining Member Functions	288
9.7.10	Member Functions vs. Global Functions	288
9.7.11	Using Class <code>Time</code>	288
9.7.12	Object Size	290
9.8	Compilation and Linking Process	290
9.9	Class Scope and Accessing Class Members	291
9.10	Access Functions and Utility Functions	292
9.11	Time Class Case Study: Constructors with Default Arguments	292
9.11.1	Class <code>Time</code>	292
9.11.2	Overloaded Constructors and C++11 Delegating Constructors	297
9.12	Destructors	298
9.13	When Constructors and Destructors Are Called	298
9.14	Time Class Case Study: A Subtle Trap —Returning a Reference or a Pointer to a <code>private</code> Data Member	302
9.15	Default Assignment Operator	304
9.16	<code>const</code> Objects and <code>const</code> Member Functions	306
9.17	Composition: Objects as Members of Classes	308
9.18	<code>friend</code> Functions and <code>friend</code> Classes	313
9.19	The <code>this</code> Pointer	314
9.19.1	Implicitly and Explicitly Using the <code>this</code> Pointer to Access an Object's Data Members	315
9.19.2	Using the <code>this</code> Pointer to Enable Cascaded Function Calls	316
9.20	<code>static</code> Class Members: Classwide Data and Member Functions	320
9.21	Aggregates in C++20	324
9.21.1	Initializing an Aggregate	325
9.21.2	C++20: Designated Initializers	325
9.22	Objects-Natural Case Study: Serialization with JSON	326
9.22.1	Serializing a vector of Objects Containing <code>public</code> Data	327
9.22.2	Serializing a vector of Objects Containing <code>private</code> Data	331
9.23	Wrap-Up	333

10	OOP: Inheritance and Runtime Polymorphism	335
10.1	Introduction	336
10.2	Base Classes and Derived Classes	339
	10.2.1 <code>CommunityMember</code> Class Hierarchy	339
	10.2.2 Shape Class Hierarchy and <code>public</code> Inheritance	340
10.3	Relationship Between Base and Derived Classes	341
	10.3.1 Creating and Using a <code>SalariedEmployee</code> Class	341
	10.3.2 Creating a <code>SalariedEmployee</code> – <code>SalariedCommissionEmployee</code> Inheritance Hierarchy	344
10.4	Constructors and Destructors in Derived Classes	349
10.5	Intro to Runtime Polymorphism: Polymorphic Video Game	350
10.6	Relationships Among Objects in an Inheritance Hierarchy	351
	10.6.1 Invoking Base-Class Functions from Derived-Class Objects	352
	10.6.2 Aiming Derived-Class Pointers at Base-Class Objects	354
	10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers	355
10.7	Virtual Functions and Virtual Destructors	357
	10.7.1 Why <code>virtual</code> Functions Are Useful	357
	10.7.2 Declaring <code>virtual</code> Functions	357
	10.7.3 Invoking a <code>virtual</code> Function	357
	10.7.4 <code>virtual</code> Functions in the <code>SalariedEmployee</code> Hierarchy	358
	10.7.5 <code>virtual</code> Destructors	361
	10.7.6 <code>final</code> Member Functions and Classes	361
10.8	Abstract Classes and Pure <code>virtual</code> Functions	362
	10.8.1 Pure <code>virtual</code> Functions	363
	10.8.2 Device Drivers: Polymorphism in Operating Systems	363
10.9	Case Study: Payroll System Using Runtime Polymorphism	363
	10.9.1 Creating Abstract Base Class <code>Employee</code>	364
	10.9.2 Creating Concrete Derived Class <code>SalariedEmployee</code>	367
	10.9.3 Creating Concrete Derived Class <code>CommissionEmployee</code>	368
	10.9.4 Demonstrating Runtime Polymorphic Processing	370
10.10	Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”	373
10.11	Non-Virtual Interface (NVI) Idiom	376
10.12	Program to an Interface, Not an Implementation	383
	10.12.1 Rethinking the Employee Hierarchy— <code>CompensationModel</code> Interface	385
	10.12.2 Class <code>Employee</code>	385
	10.12.3 <code>CompensationModel</code> Implementations	387
	10.12.4 Testing the New Hierarchy	389
	10.12.5 Dependency Injection Design Benefits	390
10.13	Runtime Polymorphism with <code>std::variant</code> and <code>std::visit</code>	391
10.14	Multiple Inheritance	397
	10.14.1 Diamond Inheritance	401
	10.14.2 Eliminating Duplicate Subobjects with <code>virtual</code> Base-Class Inheritance	403
10.15	<code>protected</code> Class Members: A Deeper Look	405

10.16	<code>public</code> , <code>protected</code> and <code>private</code> Inheritance	406
10.17	More Runtime Polymorphism Techniques; Compile-Time Polymorphism	408
	10.17.1 Other Runtime Polymorphism Techniques	408
	10.17.2 Compile-Time (Static) Polymorphism Techniques	410
	10.17.3 Other Polymorphism Concepts	411
10.18	Wrap-Up	412

11 Operator Overloading, Copy/Move Semantics and Smart Pointers **415**

11.1	Introduction	416
11.2	Using the Overloaded Operators of Standard Library Class <code>string</code>	418
11.3	Operator Overloading Fundamentals	423
	11.3.1 Operator Overloading Is Not Automatic	423
	11.3.2 Operators That Cannot Be Overloaded	423
	11.3.3 Operators That You Do Not Have to Overload	424
	11.3.4 Rules and Restrictions on Operator Overloading	424
11.4	(Downplaying) Dynamic Memory Management with <code>new</code> and <code>delete</code>	425
11.5	Modern C++ Dynamic Memory Management: RAII and Smart Pointers	427
	11.5.1 Smart Pointers	427
	11.5.2 Demonstrating <code>unique_ptr</code>	428
	11.5.3 <code>unique_ptr</code> Ownership	429
	11.5.4 <code>unique_ptr</code> to a Built-In Array	430
11.6	<code>MyArray</code> Case Study: Crafting a Valuable Class with Operator Overloading	430
	11.6.1 Special Member Functions	431
	11.6.2 Using Class <code>MyArray</code>	432
	11.6.3 <code>MyArray</code> Class Definition	441
	11.6.4 Constructor That Specifies a <code>MyArray</code> 's Size	442
	11.6.5 C++11 Passing a Braced Initializer to a Constructor	443
	11.6.6 Copy Constructor and Copy Assignment Operator	444
	11.6.7 Move Constructor and Move Assignment Operator	447
	11.6.8 Destructor	450
	11.6.9 <code>toString</code> and <code>size</code> Functions	451
	11.6.10 Overloading the Equality (<code>==</code>) and Inequality (<code>!=</code>) Operators	451
	11.6.11 Overloading the Subscript (<code>[]</code>) Operator	453
	11.6.12 Overloading the Unary <code>bool</code> Conversion Operator	454
	11.6.13 Overloading the Preincrement Operator	454
	11.6.14 Overloading the Postincrement Operator	455
	11.6.15 Overloading the Addition Assignment Operator (<code>+=</code>)	456
	11.6.16 Overloading the Binary Stream Extraction (<code>>></code>) and Stream Insertion (<code><<</code>) Operators	456
	11.6.17 <code>friend</code> Function <code>swap</code>	459
11.7	C++20 Three-Way Comparison Operator (<code><=></code>)	459
11.8	Converting Between Types	462
11.9	<code>explicit</code> Constructors and Conversion Operators	463
11.10	Overloading the Function Call Operator (<code>()</code>)	466
11.11	Wrap-Up	466

12 Exceptions and a Look Forward to Contracts 467

12.1	Introduction	468
12.2	Exception-Handling Flow of Control	471
	12.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur	472
	12.2.2 Demonstrating Exception Handling	472
	12.2.3 Enclosing Code in a try Block	474
	12.2.4 Defining a catch Handler for <code>DivideByZeroExceptions</code>	474
	12.2.5 Termination Model of Exception Handling	475
	12.2.6 Flow of Control When the User Enters a Nonzero Denominator	476
	12.2.7 Flow of Control When the User Enters a Zero Denominator	476
12.3	Exception Safety Guarantees and <code>noexcept</code>	476
12.4	Rethrowing an Exception	477
12.5	Stack Unwinding and Uncaught Exceptions	479
12.6	When to Use Exception Handling	481
	12.6.1 <code>assert</code> Macro	483
	12.6.2 Failing Fast	483
12.7	Constructors, Destructors and Exception Handling	483
	12.7.1 Throwing Exceptions from Constructors	484
	12.7.2 Catching Exceptions in Constructors via Function try Blocks	484
	12.7.3 Exceptions and Destructors: Revisiting <code>noexcept(false)</code>	486
12.8	Processing new Failures	487
	12.8.1 new Throwing <code>bad_alloc</code> on Failure	488
	12.8.2 new Returning <code>nullptr</code> on Failure	489
	12.8.3 Handling new Failures Using Function <code>set_new_handler</code>	489
12.9	Standard Library Exception Hierarchy	490
12.10	C++'s Alternative to the <code>finally</code> Block: Resource Acquisition Is Initialization (RAII)	493
12.11	Some Libraries Support Both Exceptions and Error Codes	493
12.12	Logging	494
12.13	Looking Ahead to Contracts	495
12.14	Wrap-Up	503

13 Standard Library Containers and Iterators 505

13.1	Introduction	506
13.2	Introduction to Containers	508
	13.2.1 Common Nested Types in Sequence and Associative Containers	510
	13.2.2 Common Container Member and Non-Member Functions	510
	13.2.3 Requirements for Container Elements	513
13.3	Working with Iterators	513
	13.3.1 Using <code>istream_iterator</code> for Input and <code>ostream_iterator</code> for Output	514
	13.3.2 Iterator Categories	515
	13.3.3 Container Support for Iterators	516

13.3.4	Predefined Iterator Type Names	516
13.3.5	Iterator Operators	516
13.4	A Brief Introduction to Algorithms	518
13.5	Sequence Containers	518
13.6	vector Sequence Container	519
13.6.1	Using vectors and Iterators	519
13.6.2	vector Element-Manipulation Functions	522
13.7	list Sequence Container	526
13.8	deque Sequence Container	531
13.9	Associative Containers	533
13.9.1	multiset Associative Container	533
13.9.2	set Associative Container	537
13.9.3	multimap Associative Container	539
13.9.4	map Associative Container	541
13.10	Container Adaptors	543
13.10.1	stack Adaptor	543
13.10.2	queue Adaptor	545
13.10.3	priority_queue Adaptor	546
13.11	bitset Near Container	547
13.12	Optional: A Brief Intro to Big O	549
13.13	Optional: A Brief Intro to Hash Tables	552
13.14	Wrap-Up	553

14 Standard Library Algorithms and C++20 Ranges & Views **555**

14.1	Introduction	556
14.2	Algorithm Requirements: C++20 Concepts	558
14.3	Lambdas and Algorithms	560
14.4	Algorithms	563
14.4.1	fill, fill_n, generate and generate_n	563
14.4.2	equal, mismatch and lexicographical_compare	566
14.4.3	remove, remove_if, remove_copy and remove_copy_if	568
14.4.4	replace, replace_if, replace_copy and replace_copy_if	572
14.4.5	Shuffling, Counting, and Minimum and Maximum Element Algorithms	574
14.4.6	Searching and Sorting Algorithms	578
14.4.7	swap, iter_swap and swap_ranges	582
14.4.8	copy_backward, merge, unique, reverse, copy_if and copy_n	584
14.4.9	inplace_merge, unique_copy and reverse_copy	588
14.4.10	Set Operations	589
14.4.11	lower_bound, upper_bound and equal_range	592
14.4.12	min, max and minmax	594
14.4.13	Algorithms gcd, lcm, iota, reduce and partial_sum from Header <numeric>	596
14.4.14	Heapsort and Priority Queues	599

14.5	Function Objects (Functors)	603
14.6	Projections	608
14.7	C++20 Views and Functional-Style Programming	611
14.7.1	Range Adaptors	611
14.7.2	Working with Range Adaptors and Views	612
14.8	Intro to Parallel Algorithms	617
14.9	Standard Library Algorithm Summary	619
14.10	A Look Ahead to C++23 Ranges	622
14.11	Wrap-Up	623

15 Templates, C++20 Concepts and Metaprogramming **625**

15.1	Introduction	626
15.2	Custom Class Templates and Compile-Time Polymorphism	629
15.3	C++20 Function Template Enhancements	634
15.3.1	C++20 Abbreviated Function Templates	634
15.3.2	C++20 Templated Lambdas	636
15.4	C++20 Concepts: A First Look	636
15.4.1	Unconstrained Function Template <code>multiply</code>	637
15.4.2	Constrained Function Template with a C++20 Concepts <code>requires</code> Clause	640
15.4.3	C++20 Predefined Concepts	642
15.5	Type Traits	644
15.6	C++20 Concepts: A Deeper Look	648
15.6.1	Creating a Custom Concept	648
15.6.2	Using a Concept	649
15.6.3	Using Concepts in Abbreviated Function Templates	650
15.6.4	Concept-Based Overloading	651
15.6.5	<code>requires</code> Expressions	654
15.6.6	C++20 Exposition-Only Concepts	657
15.6.7	Techniques Before C++20 Concepts: SFINAE and Tag Dispatch	658
15.7	Testing C++20 Concepts with <code>static_assert</code>	659
15.8	Creating a Custom Algorithm	661
15.9	Creating a Custom Container and Iterators	663
15.9.1	Class Template <code>ConstIterator</code>	665
15.9.2	Class Template <code>Iterator</code>	668
15.9.3	Class Template <code>MyArray</code>	670
15.9.4	<code>MyArray</code> Deduction Guide for Braced Initialization	673
15.9.5	Using <code>MyArray</code> and Its Custom Iterators with <code>std::ranges</code> Algorithms	674
15.10	Default Arguments for Template Type Parameters	678
15.11	Variable Templates	678
15.12	Variadic Templates and Fold Expressions	679
15.12.1	tuple Variadic Class Template	679

15.12.2	Variadic Function Templates and an Intro to C++17 Fold Expressions	682
15.12.3	Types of Fold Expressions	686
15.12.4	How Unary-Fold Expressions Apply Their Operators	686
15.12.5	How Binary-Fold Expressions Apply Their Operators	689
15.12.6	Using the Comma Operator to Repeatedly Perform an Operation	690
15.12.7	Constraining Parameter Pack Elements to the Same Type	691
15.13	Template Metaprogramming	693
15.13.1	C++ Templates Are Turing Complete	694
15.13.2	Computing Values at Compile-Time	694
15.13.3	Conditional Compilation with Template Metaprogramming and <code>constexpr if</code>	699
15.13.4	Type Metafunctions	701
15.14	Wrap-Up	705

16 C++20 Modules: Large-Scale Development **707**

16.1	Introduction	708
16.2	Compilation and Linking Before C++20	710
16.3	Advantages and Goals of Modules	711
16.4	Example: Transitioning to Modules—Header Units	712
16.5	Modules Can Reduce Translation Unit Sizes and Compilation Times	715
16.6	Example: Creating and Using a Module	716
16.6.1	<code>module</code> Declaration for a Module Interface Unit	717
16.6.2	Exporting a Declaration	719
16.6.3	Exporting a Group of Declarations	719
16.6.4	Exporting a namespace	719
16.6.5	Exporting a namespace Member	720
16.6.6	Importing a Module to Use Its Exported Declarations	720
16.6.7	Example: Attempting to Access Non-Exported Module Contents	722
16.7	Global Module Fragment	724
16.8	Separating Interface from Implementation	725
16.8.1	Example: Module Implementation Units	725
16.8.2	Example: Modularizing a Class	728
16.8.3	<code>:private</code> Module Fragment	731
16.9	Partitions	732
16.9.1	Example: Module Interface Partition Units	732
16.9.2	Module Implementation Partition Units	735
16.9.3	Example: “Submodules” vs. Partitions	736
16.10	Additional Modules Examples	740
16.10.1	Example: Importing the C++ Standard Library as Modules	740
16.10.2	Example: Cyclic Dependencies Are Not Allowed	742
16.10.3	Example: <code>imports</code> Are Not Transitive	743
16.10.4	Example: Visibility vs. Reachability	744
16.11	Migrating Code to Modules	746

16.12	Future of Modules and Modules Tooling	746
16.13	Wrap-Up	748

17 Parallel Algorithms and Concurrency: A High-Level View **755**

17.1	Introduction	756
17.2	Standard Library Parallel Algorithms (C++17)	759
17.2.1	Example: Profiling Sequential and Parallel Sorting Algorithms	759
17.2.2	When to Use Parallel Algorithms	762
17.2.3	Execution Policies	763
17.2.4	Example: Profiling Parallel and Vectorized Operations	764
17.2.5	Additional Parallel Algorithm Notes	766
17.3	Multithreaded Programming	767
17.3.1	Thread States and the Thread Life Cycle	767
17.3.2	Deadlock and Indefinite Postponement	769
17.4	Launching Tasks with <code>std::jthread</code>	771
17.4.1	Defining a Task to Perform in a Thread	772
17.4.2	Executing a Task in a <code>jthread</code>	773
17.4.3	How <code>jthread</code> Fixes <code>thread</code>	775
17.5	Producer–Consumer Relationship: A First Attempt	776
17.6	Producer–Consumer: Synchronizing Access to Shared Mutable Data	783
17.6.1	Class <code>SynchronizedBuffer</code> : Mutexes, Locks and Condition Variables	785
17.6.2	Testing <code>SynchronizedBuffer</code>	791
17.7	Producer–Consumer: Minimizing Waits with a Circular Buffer	795
17.8	Readers and Writers	804
17.9	Cooperatively Canceling <code>jthreads</code>	805
17.10	Launching Tasks with <code>std::async</code>	808
17.11	Thread-Safe, One-Time Initialization	815
17.12	A Brief Introduction to Atomics	816
17.13	Coordinating Threads with C++20 Latches and Barriers	820
17.13.1	C++20 <code>std::latch</code>	820
17.13.2	C++20 <code>std::barrier</code>	823
17.14	C++20 Semaphores	826
17.15	C++23: A Look to the Future of C++ Concurrency	830
17.15.1	Parallel Ranges Algorithms	830
17.15.2	Concurrent Containers	830
17.15.3	Other Concurrency-Related Proposals	831
17.16	Wrap-Up	831

18 C++20 Coroutines **833**

18.1	Introduction	834
18.2	Coroutine Support Libraries	835
18.3	Installing the <code>conurrencpp</code> and <code>generator</code> Libraries	837

Preface



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

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

P.1 Modern C++

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

Keep It Topical

“*Who dares to teach must never cease to learn.*”⁴ (J. C. Dana)

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

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

C++ Versions

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

P.2 Target Audiences

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

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

P.3 Live-Code Approach and Getting the Code

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

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

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

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

deitel@deitel.com

P.4 Three Industrial-Strength Compilers

We tested the code examples on the latest versions of

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

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

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

<https://github.com/pdeitel/CPlusPlus20ForProgrammers>

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

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

<https://informat.com/title/9780136905691>

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

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

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

P.6 “Objects-Natural” Learning Approach

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

What Is Objects Natural?

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

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

Free Classes

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

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

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

The Boost Project

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

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

Objects-Natural Case Studies

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

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

5. “Boost 1.78.0 Library Documentation.” Accessed January 9, 2022. https://www.boost.org/doc/libs/1_78_0/.

6. “Boost C++ Libraries.” Wikipedia. Wikimedia Foundation. Accessed January 9, 2022. [https://en.wikipedia.org/wiki/Boost_\(C%2B%2B_libraries\)](https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries)).

7. Kennytm, Answer to “Which Boost Features Overlap with C++11?” Accessed January 9, 2022. <https://stackoverflow.com/a/8852421>.

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

P.7 A Tour of the Book

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

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

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

Part I: Programming Fundamentals Quickstart

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

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

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

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

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

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

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

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

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

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

Sec 
11

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

Part 2: Arrays, Pointers and Strings

20

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. We begin our introduction to **functional-style programming with lambda expressions** (anonymous functions) and C++20’s **Ranges**—one of C++20’s “big four” features. **Section 6.15’s objects-natural case study** demonstrates the C++ standard library class **template `vector`**. **This entire chapter is essentially a large objects-natural case study of both arrays and vectors**. The code in this chapter is a good example of Modern C++ coding idioms.

8. Fred Long, “Do Not Use the `rand()` Function for Generating Pseudorandom Numbers.” Last modified by Jill Britton on November 20, 2021. Accessed December 27, 2021. <https://wiki.sei.cmu.edu/confluence/display/c/MS30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers>.

Chapter 7, (Downplaying) Pointers in Modern C++, provides thorough coverage of pointers and the intimate relationship among built-in pointers, pointer-based arrays and pointer-based strings (also called C-strings), each of which C++ inherited from the C programming language. Pointers are powerful but challenging to work with and are error-prone. So, we point out Modern C++ features that **eliminate the need for most pointers** and make your code more robust and secure, including **arrays** and **vectors**, **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 code. **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. This chapter again emphasizes Modern C++ coding idioms.

 Sec
20
17

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 datasets); and introduces string pattern matching with the standard library's regular-expression (regex) capabilities. C++ offers *two* types of strings—**string** objects and **C-style pointer-based strings**. We use **string** class objects to make programs more robust and **eliminate many of the security problems of C strings**. **In new development, you should favor string objects.** We also present C++17's **string_views**—a lightweight, flexible mechanism for passing any type of string to a function. This chapter presents **two objects-natural case studies**:

 Sec
17

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

Part 3: Object-Oriented Programming

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

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

Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, shows how to enable C++’s existing operators to work with custom class objects, and introduces smart pointers and **dynamic memory management**. Smart pointers help you avoid dynamic memory management errors by providing additional functionality beyond that of built-in pointers. We discuss **unique_ptr** in this chapter and **shared_ptr** and **weak_ptr** in online Chapter 20. A key aspect of this chapter is crafting valuable classes. We begin with a **string class test-drive**, presenting an elegant use of operator overloading before you implement your own customized class with overloaded operators. Then, in one of the book’s most important case studies, you’ll build your own custom `MyArray` class using overloaded operators and other capabilities to **solve various problems with C++’s native pointer-based arrays**.⁹ We introduce and implement the five **special member functions** you can define in each class—the **copy constructor**, **copy assignment operator**, **move constructor**, **move assignment operator** and **destructor**. We discuss **copy semantics** and **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 `MyArray` to a class template that can store elements of a specified type. You will have truly crafted valuable classes.

Err Perf 
20

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

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 accessing container elements) and
- **algorithms** (which use iterators to manipulate containers).

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

20 C++20 **Ranges** can simplify your code.

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

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

Part 5: Advanced Topics

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

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

Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, is one of the most important chapters in the book, presenting C++’s features for building applications that create and manage **multiple tasks**. This can significantly improve program performance and responsiveness. We show how to use C++17’s **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 C++’s `<chrono>` library features to profile the performance improvement we get on today’s popular multi-core systems, as we employ an increasing number of cores. You’ll see that the parallel sort runs 6.76 times faster than the Perf 17

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

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

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



Appendices

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

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

P.8 How to Get the Online Chapters and Appendices

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

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

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

Online Chapters

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

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

Online Appendices

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

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

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

Web-Based Materials on deitel.com

Our deitel.com web page for the book

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

contains the following additional resources:

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

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

P.9 C++ Core Guidelines

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

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

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

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

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



11. C++ Core Guidelines, “Abstract.” Accessed January 9, 2020. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract>.

12. C++ Core Guidelines, “Abstract.”

Guidelines Support Library

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

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

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

P.10 Industrial-Strength Static Code Analysis Tools



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

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

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

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



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

P.11 Teaching Approach



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

13. C++ Core Guidelines, “GSL: Guidelines Support Library.” Accessed January 9, 2022. <https://iso-cpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl>.

Using Fonts for Emphasis

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

Syntax Coloring

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

```

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

```

Objectives and Outline

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

Tables and Illustrations

Abundant tables and line drawings are included.

Programming Tips and Key Features

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

Index

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

P.12 Developer Resources

StackOverflow

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

GitHub

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

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

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

15. “GitHub.” Accessed January 7, 2022. <https://github.com/>.

code written in an enormous number of programming languages¹⁶—developers contributed to 61+ million repositories in the last year.¹⁷ **GitHub** is a crucial element of the professional software developer’s arsenal with **version-control tools** that help developer teams manage public open-source projects and private projects.

Sec Sec 

There is a massive C++ open-source community. On GitHub, there are over 41,000¹⁸ C++ code repositories. You can check out other people’s C++ code on GitHub and even build upon it if you like. This is a great way to learn and is a natural extension of our live-code teaching approach.¹⁹

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

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

P.13 Docker

Sec 

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

We show how to install and execute Docker containers preconfigured with

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

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

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

P.14 Some Key C++ Documentation and Resources

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

16. “Where the World Builds Software.” Accessed January 7, 2022. <https://github.com/about>.

17. “The 2021 State of the Octoverse.” Accessed January 7, 2022. <https://octoverse.github.com>.

18. “C++.” Accessed January 7, 2022. <https://github.com/topics/cpp>.

19. Students will need to become familiar with the variety of open-source licenses for software on GitHub.

20. “Microsoft to Acquire GitHub for \$7.5 Billion.” Accessed January 7, 2022. <https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/>.

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

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

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

Documentation

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

Blogs

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

Additional Resources

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

P.15 Getting Your Questions Answered

Popular C++ and general programming online forums include

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

For a list of other valuable sites, see

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



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

Communicating with the Authors

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

deitel@deitel.com

We'll respond promptly.

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

Join the Deitel social media communities on

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

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

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

<https://learning.oreilly.com/register/>

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

<https://deitel.com/LearnWithDeitel>

Textbooks and Professional Books

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

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

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

Asynchronous LiveLessons Video Products

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

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

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

Live Full-Throttle Training Courses

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

<https://deitel.com/LearnWithDeitel>

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

P.18 Live Instructor-Led Training with Paul Deitel

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

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

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

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

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

- A chapter introducing hardware, software and Internet concepts.

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

Deitel Pearson eTexts and Revels include:

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

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

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

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

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

P.20 Acknowledgments

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

<https://learning.oreilly.com/>

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

Reviewers

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

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

- 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* 17 20 23
- Saar Raz, Senior Software Engineer, Swimm.io and Implementor of C++20 Concepts in Clang 20
- José Antonio González Seco, Parliament of Andalusia
- Anthony Williams, Member of the British Standards Institution C++ Standards Panel, Director of Just Software Solutions Ltd., Author of C++ *Concurrency in Action, 2/e* (Anthony is the author or co-author of many C++ Standard Committee papers that led to C++’s standardized concurrency features)

Arthur O’Dwyer

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

GitHub

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

Matt Godbolt and Compiler Explorer

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

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

21. “Where the World Builds Software.” Accessed January 7, 2022. <https://github.com/about>.

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

Dietmar Kühl

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

Rainer Grimm

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

Brian Goetz

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

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

Open-Source Contributors and Bloggers

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

Google Search

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

Grammarly

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

22. “Grammarly.” Accessed January 15, 2022. <https://www.grammarly.com>.

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

<https://www.grammarly.com/plans>

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

deitel@deitel.com

We’ll respond promptly.

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

Paul Deitel
Harvey Deitel

About the Authors

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

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

23. “Our Mission.” Accessed January 15, 2022. <https://www.grammarly.com/about>.

About Deitel® & Associates, Inc.

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

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

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

deitel@deitel.com

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

<https://deitel.com/training>

Individuals wishing to purchase Deitel books can do so at

<https://amazon.com>

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

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

corpsales@pearsoned.com

Deitel e-books are available in various formats from

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

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

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

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

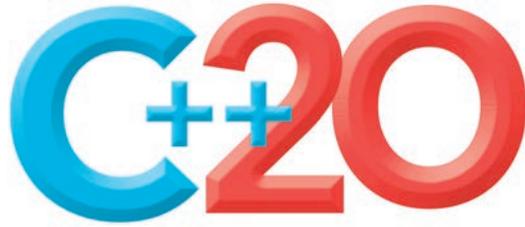
<https://www.informit.com/>

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

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

<https://learning.oreilly.com/register/>

Before You Begin



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

Font and Naming Conventions

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

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

Obtaining the Code Examples

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

<https://deitel.com/cpp20fp>

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

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

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

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

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

- For Microsoft Windows, we used Microsoft Visual Studio Community edition, which includes the Visual C++ compiler and other Microsoft development tools.¹
- For macOS, we used the Apple Xcode² C++ compiler, which uses a version of the Clang C++ compiler.
- For Linux, we used the GNU C++ compiler³—part of the GNU Compiler Collection (GCC). GNU C++ is already installed on most Linux systems (though

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

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

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

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

Some Examples Do Not Compile and Run on All Three Compilers

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

<https://deitel.com/cpp20fp>

Installing Visual Studio Community Edition on Windows

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

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

Next, go to

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

Then perform the following installation steps:

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

Installing Xcode on macOS

On macOS, perform the following steps to install Xcode:

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

Installing the Most Recent GNU C++ Version

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

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

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

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

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

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

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

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

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

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

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

Docker and Docker Containers

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

with new technologies quickly and conveniently. It is also a great way to experiment with new compiler versions.

Installing Docker

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

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

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

<https://docs.docker.com/engine/install/>

Downloading the GNU Compiler Collection Docker Container

The GNU team maintains official Docker containers at

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

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

```
docker pull gcc:latest
```

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

Downloading the GNU Compiler Collection Docker Container

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

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

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

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

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

Getting Your C++ Questions Answered

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

deitel@deitel.com

and

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

We'll respond promptly.

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

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

<https://stackoverflow.com>

on which you can

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

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

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

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

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

Online C++ Documentation

For documentation on the C++ standard library, visit

<https://cppreference.com>

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

<https://isocpp.org/faq>

A Note Regarding the `{fmt}` Text-Formatting Library

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

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

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

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

```
#include <format>
```

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

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

Static Code Analysis Tools

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

5. “`{fmt}`.” Accessed February 15, 2022. <https://github.com/fmtlib/fmt>.

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

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

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

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

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

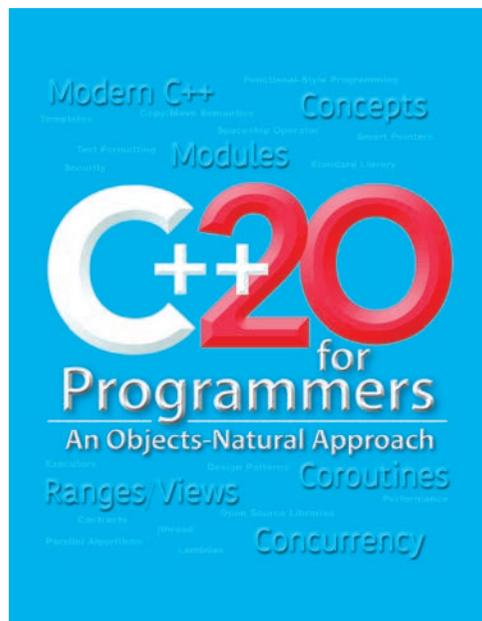
4

Control Statements, Part 2

Objectives

In this chapter, you'll:

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



4.1	Introduction	4.11	Logical Operators
4.2	Essentials of Counter-Controlled Iteration	4.11.1	Logical AND (&&) Operator
4.3	for Iteration Statement	4.11.2	Logical OR () Operator
4.4	Examples Using the for Statement	4.11.3	Short-Circuit Evaluation
4.5	Application: Summing Even Integers	4.11.4	Logical Negation (!) Operator
4.6	Application: Compound-Interest Calculations	4.11.5	Example: Producing Logical-Operator Truth Tables
4.7	do...while Iteration Statement	4.12	Confusing the Equality (==) and Assignment (=) Operators
4.8	switch Multiple-Selection Statement	4.13	Objects-Natural Case Study: Using the miniz-cpp Library to Write and Read ZIP files
4.9	C++17 Selection Statements with Initializers	4.14	C++20 Text Formatting with Field Widths and Precisions
4.10	break and continue Statements	4.15	Wrap-Up

4.1 Introduction

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

4.2 Essentials of Counter-Controlled Iteration

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

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

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

```

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

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

```

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

```

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

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

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

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

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

4.3 for Iteration Statement

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

```

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 }

```

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

```

13     cout << "\n";
14 }

```

```

1 2 3 4 5 6 7 8 9 10

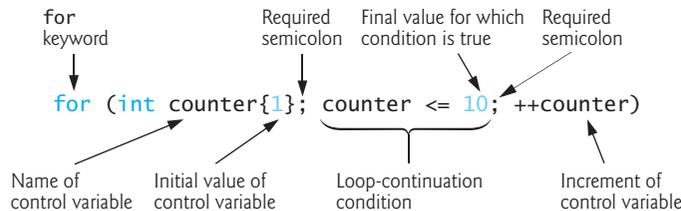
```

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

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

A Closer Look at the for Statement's Header

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



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

General Format of a for Statement

The general format of the for statement is

```

for (initialization; loopContinuationCondition; increment) {
    statement
}

```

where

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

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

Scope of a for Statement's Control Variable

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

Expressions in a for Statement's Header Are Optional

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

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

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

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

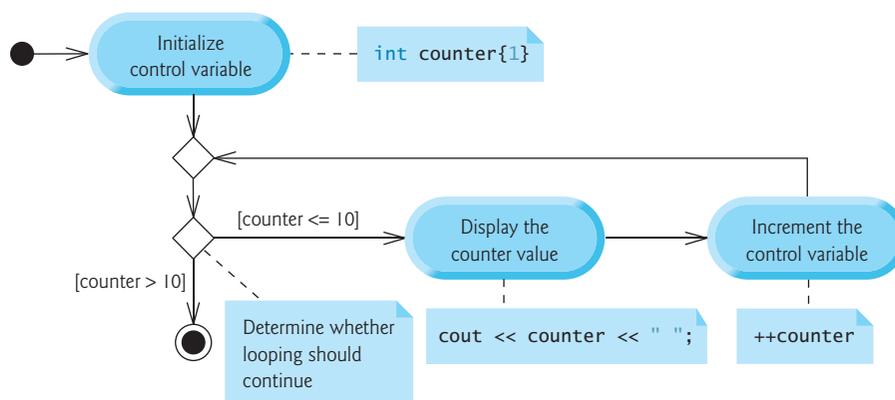
Using a for Statement's Control Variable in the Statement's Body

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



UML Activity Diagram of the for Statement

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



4.4 Examples Using the for Statement

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Application: Summing Even Integers

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

```

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

```

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

```

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

```

```
Sum is 110
```

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

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

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

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

4.6 Application: Compound-Interest Calculations

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

A person invests \$1,000 in a savings account yielding 5% interest. Assuming all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p(1 + r)^n$$

where

p is the original amount invested (i.e., the principal),
 r is the annual interest rate (e.g., use 0.05 for 5%),
 n is the number of years, and
 a is the amount on deposit at the end of the n th year.

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

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

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

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

```

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

```

```

Initial principal: 1000.00
    Interest rate:    0.05

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

```

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

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

Formatting with Field Widths and Justification

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

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

Performing the Interest Calculations with Standard Library Function `pow`

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

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

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



Floating-Point Number Precision and Memory Requirements

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

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

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

Floating-Point Numbers Are Approximations

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



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.³

A Warning about Displaying Rounded Values

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

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

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

Even Common Dollar Amounts Can Have Floating-Point Representational Errors

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

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

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

4.7 `do...while` Iteration Statement

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

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

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

```

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

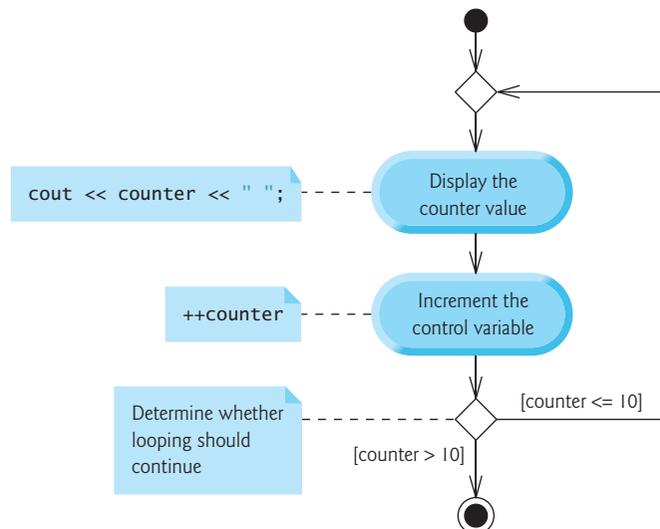
```

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

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

UML Activity Diagram for the do...while Iteration Statement

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



4.8 switch Multiple-Selection Statement

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

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

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

```

1 // fig04_06.cpp
2 // Using a switch statement to count letter grades.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main() {
8     int total{0}; // sum of grades
9     int gradeCounter{0}; // number of grades entered
10    int aCount{0}; // count of A grades
11    int bCount{0}; // count of B grades
12    int cCount{0}; // count of C grades
13    int dCount{0}; // count of D grades
14    int fCount{0}; // count of F grades
15
16    cout << "Enter the integer grades in the range 0-100.\n"
17         << "Type the end-of-file indicator to terminate input:\n"
18         << "   On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
19         << "   On Windows type <Ctrl> z then press Enter\n";
20
21    int grade;
22
23    // loop until user enters the end-of-file indicator
24    while (cin >> grade) {
25        total += grade; // add grade to total
26        ++gradeCounter; // increment number of grades
27
28        // increment appropriate letter-grade counter
29        switch (grade / 10) {
30            case 9: // grade was between 90
31                case 10: // and 100, inclusive
32                    ++aCount;
33                    break; // exits switch
34
35            case 8: // grade was between 80 and 89
36                ++bCount;
37                break; // exits switch
38

```

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

```

39         case 7: // grade was between 70 and 79
40             ++cCount;
41             break; // exits switch
42
43         case 6: // grade was between 60 and 69
44             ++dCount;
45             break; // exits switch
46
47         default: // grade was less than 60
48             ++fCount;
49             break; // optional; exits switch anyway
50     } // end switch
51 } // end while
52
53 // set floating-point number format
54 cout << fixed << setprecision(2);
55
56 // display grade report
57 cout << "\nGrade Report:\n";
58
59 // if user entered at least one grade...
60 if (gradeCounter != 0) {
61     // calculate average of all grades entered
62     double average{static_cast<double>(total) / gradeCounter};
63
64     // output summary of results
65     cout << "Total of the " << gradeCounter << " grades entered is "
66         << total << "\nClass average is " << average
67         << "\nNumber of students who received each grade:"
68         << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69         << "\nD: " << dCount << "\nF: " << fCount << "\n";
70 }
71 else { // no grades were entered, so output appropriate message
72     cout << "No grades were entered" << "\n";
73 }
74 }

```

```

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

```

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

```

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

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

```

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

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

Reading Grades from the User

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

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

```
<Ctrl> d
```

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

```
<Ctrl> z
```

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

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

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

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

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

Processing the Grades

The `switch` statement (lines 29–50) determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 rep-

represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The `switch` statement’s block contains a sequence of **case labels** and an optional **default case**, which can appear anywhere in the `switch`, but normally appears last. These are used in this example to determine which counter to increment based on the grade.

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

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

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

case without a break Statement—C++17 `[[fall through]]` Attribute

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.⁴

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

```
[[fall through]];
```

where the `break` statement would normally appear.

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

The default Case

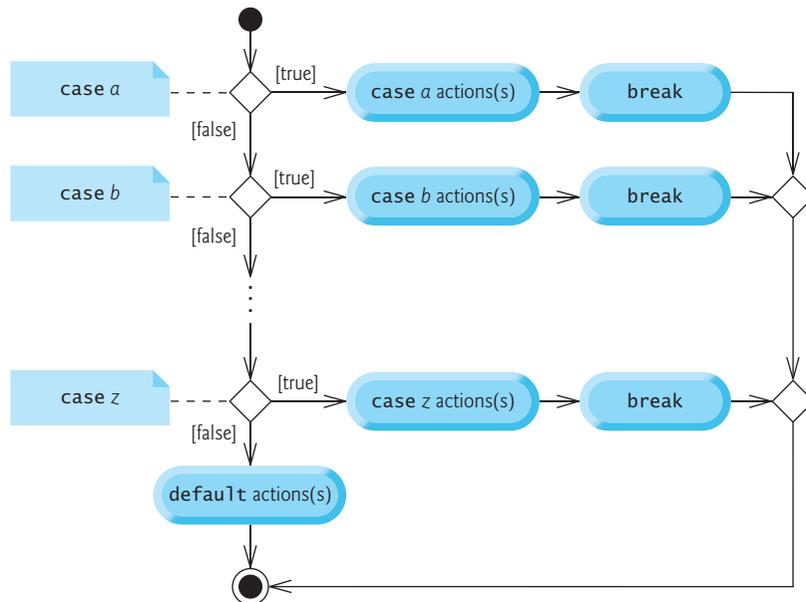
If no match occurs between the controlling expression's value and any of the case labels, the default case (lines 47–49) executes. We use the default case in this example to process all controlling-expression values that are less than 6—that is, all failing grades. If no match occurs and the switch does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

Displaying the Grade Report

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

switch Statement UML Activity Diagram

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



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

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

Notes on cases

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

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

4.9 C++17 Selection Statements with Initializers 17

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

```

1 // fig04_07.cpp
2 // C++17 if statements with initializers.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     if (int value{7}; value == 7) {
8         cout << "value is " << value << "\n";
9     }
10    else {
11        cout << "value is not 7; it is " << value << "\n";
12    }
13
14    if (int value{13}; value == 9) {
15        cout << "value is " << value << "\n";
16    }
17    else {
18        cout << "value is not 9; it is " << value << "\n";
19    }
20 }
```

```

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

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

Syntax of Selection Statements with Initializers

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

Scope of Variables Declared in the Initializer

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

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

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

4.10 break and continue Statements

In addition to selection and iteration statements, C++ provides `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        }

```

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

```

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

```

```

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

```

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

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

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

continue Statement

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

```

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.

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

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

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

Perf 

4.11 Logical Operators

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

4.11.1 Logical AND (&&) Operator

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

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

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

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

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

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

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

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

4.1.1.2 Logical OR (||) Operator

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

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

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

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

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

expression1	expression2	expression1 expression2
false	false	false
false	true	true
true	false	true
true	true	true

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

4.1.1.3 Short-Circuit Evaluation

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

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

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

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

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

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

4.11.4 Logical Negation (!) Operator

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

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

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

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

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

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

expression	!expression
false	true
true	false

4.11.5 Example: Producing Logical-Operator Truth Tables

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

```

1 // fig04_10.cpp
2 // Logical operators.
3 #include <iostream>
4 #include <fmt/format.h> // in C++20, this will be #include <format>
5 using namespace std;
6 using namespace fmt; // not needed in C++20
7
8 int main() {
9     // create truth table for && (logical AND) operator
10    cout << "Logical AND (&&)\n"
11        << format("false && false: {}\n", false && false)
12        << format("false && true: {}\n", false && true)
13        << format("true && false: {}\n", true && false)
14        << format("true && true: {}\n\n", true && true);
15
16    // create truth table for || (logical OR) operator
17    cout << "Logical OR (||)\n"
18        << format("false || false: {}\n", false || false)
19        << format("false || true: {}\n", false || true)
20        << format("true || false: {}\n", true || false)
21        << format("true || true: {}\n\n", true || true);
22
23    // create truth table for ! (logical negation) operator
24    cout << "Logical negation (!)\n"
25        << format("!false: {}\n", !false)
26        << format("!true: {}\n", !true);
27 }

```

```

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

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

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

```

Fig. 4.10 | Logical operators.

Precedence and Grouping of the Operators Presented So Far

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

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

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

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

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

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

but we accidentally write

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

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

lvalues and *rvalues*

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

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

Using `==` in Place of `=`

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

```
x = 1;
```

but instead write

```
x == 1;
```

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

Enabling Warnings

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

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

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

4.13 Objects-Natural Case Study: Using the `miniz-cpp` Library to Write and Read ZIP files⁷



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

ZIP Files

You’ve probably used ZIP files—if not, you almost certainly will. The **ZIP file format**⁸ is a lossless compression⁹ format that has been in use for over 30 years. Lossless compression algorithms use various techniques for compressing data—such as

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

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

Open-Source `miniz-cpp` Library

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

7. This example does not compile in GNU C++.

8. “Zip (file format).” Accessed November 19, 2021. [https://en.wikipedia.org/wiki/Zip_\(file_format\)](https://en.wikipedia.org/wiki/Zip_(file_format)).

9. “Data compression.” Accessed November 19, 2021. https://en.wikipedia.org/wiki/Data_compression#Lossless.

10. “List of archive formats.” Wikipedia. Wikimedia Foundation, March 19, 2020. https://en.wikipedia.org/wiki/List_of_archive_formats.

11. <https://github.com/tfussell/miniz-cpp>.

12. The `miniz-cpp` library provides capabilities nearly identical to the Python standard library’s `zipfile` module (<https://docs.python.org/3/library/zipfile.html>), so the `miniz-cpp` GitHub repository refers you to that documentation page for the list of features.

```

1 // fig04_11.cpp
2 // Using the miniz-cpp header-only library to write and read a ZIP file.
3 #include <iostream>
4 #include <string>
5 #include "zip_file.hpp"
6 using namespace std;
7

```

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

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

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

```

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

```

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

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

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

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

```

13 // string literals separated only by whitespace are combined
14 // into a single string by the compiler
15 string content{
16     "This chapter introduces all but one of the remaining control "
17     "statements--the for, do...while, switch, break and continue "
18     "statements. We explore the essentials of counter-controlled "
19     "iteration. We use compound-interest calculations to begin "
20     "investigating the issues of processing monetary amounts. First, "
21     "we discuss the representational errors associated with "
22     "floating-point types. We use a switch statement to count the "
23     "number of A, B, C, D and F grade equivalents in a set of "
24     "numeric grades. We show C++17's enhancements that allow you to "
25     "initialize one or more variables of the same type in the "
26     "headers of if and switch statements."};
27

```

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

Displaying the Name and Contents of the ZIP File

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

```

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

```

```

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

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

```

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

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

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

```

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

```

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

```

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

```

```

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

```

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

Extracting "intro.txt" and Displaying Its Original Contents

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

```
50 // original file contents
51 string extractedContent{input.read(info)};
52
```

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

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

Original contents of intro.txt:

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

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

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

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

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

```

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

```

```

Initial principal: 1000.00
    Interest rate:   0.05

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

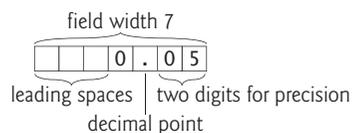
```

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

Formatting the Principal and Interest Rate

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

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



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

Formatting the Year and Amount-on-Deposit Column Heads

In line 17's format string

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

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

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

The format string in line 22

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

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

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

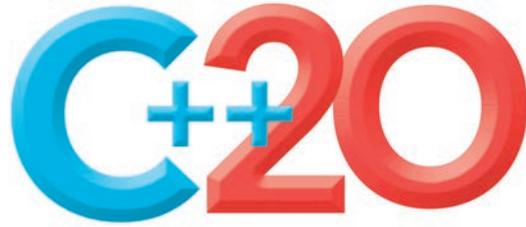
4.15 Wrap-Up

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

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

This page intentionally left blank

Index



Symbols

- 836
- , (comma operator) **75**
- : in inheritance **345**
- :: (scope resolution operator) **133, 287, 321**
- ! (logical negation) **88, 90**
- != (inequality operator) **31, 32**
- ?: (ternary conditional operator) **47, 145**
- . (member selection operator) **291, 292**
- ' (digit separator, C++14) **63**
- '\0' (null character) **216**
- '\n' (newline character) **216**
- [] (operator for map) **541**
- [] (regex character class) **262**
- [&] (lambda introducer, capture by reference) **562**
- [=] (lambda introducer, capture by value) **562**
- {n,} (quantifier in regex) **264**
- {n,m} (quantifier in regex) **264**
- * (multiplication operator) **30**
- * (pointer dereference or indirection operator) **193, 194**
- * (quantifier in regex) **263**
- *= (multiplication assignment operator) **57**
- / (division operator) **30**
- /* */ (multiline comment) **23**
- // (single-line comment) **23**
- /= (division assignment operator) **57**
- \ (regex metacharacter) **261**
- \' (single-quote-character escape sequence) **25**
- \" (double-quote-character escape sequence) **25**
- \\ (backslash-character escape sequence) **25**
- \a (alert escape sequence) **25**
- \D (regex character class) **262**
- \d (regex character class) **262, 262**
- \n (newline escape sequence) **25**
- \r (carriage-return escape sequence) **25**
- \S (regex character class) **262**
- \s (regex character class) **262**
- \t (tab escape sequence) **25**
- \W (regex character class) **262**
- \w (regex character class) **262**
- & (address operator) **193, 194**
- & (to declare reference) **129**
- && (logical AND operator) **88, 89, 145**
- &= (bitwise AND assignment operator) **548**
- % (remainder operator) **30**
- %= (remainder assignment operator) **57**
- ^ (regex metacharacter) **263**
- ^= (bitwise exclusive OR assignment operator) **549**
- + (addition operator) **29, 30**
- + (quantifier in regex) **263**
- (postfix decrement operator) **58**
- ++ (postfix increment operator) **58**
 - on an iterator **513**
- (prefix decrement operator) **58**
- ++ (prefix increment operator) **58**
 - on an iterator **513**
- += (addition assignment operator) **57**
 - string concatenation **224**
- < (less-than operator) **31**
- << (stream insertion operator) **24, 30**
- <= (less-than-or-equal-to operator) **31**
- <=> (three-way comparison operator) **418, 459, 460, 511**
- = (assignment operator) **29, 30, 304, 424, 513**
- = (subtraction assignment operator) **57**
- = 0 (pure specifier for a pure virtual function) **363**
- == (equality operator) **31, 32**
- > (arrow member selection) **291**
- > (greater-than operator) **31**
- > (in a compound C++20 concept requirement) **656**
- >= (greater-than-or-equal-to operator) **31**
- >> (stream extraction operator) **29**
- | (operator in a C++20 range pipeline) **178**
- |= (bitwise inclusive OR assignment operator) **548**
- || (logical OR operator) **88, 89, 145**

A

- abbreviated function template (C++20) 137, 634, **634**, 650, 651, 662
 - constrained auto 650
- abort standard library function **299**, 480, 489
- absolute value 104
- abstract class **362**, 363, 374
 - Employee 364
 - pure **363**
- access a global variable 133
- access function **292**
- access non-`static` class data members and member functions 324
- access privileges 202, 204
- access shared data 783
- access specifier **274**, 275, 313
 - `private` 274
 - `public` 274
- access the caller's data 129
- access violation 508
- accounts-receivable system 240
- accumulate algorithm **174**, 596, 605, 608, 621
- acquire
 - a lock **787**
 - a semaphore 827
- acquire member function of `std::binary_semaphore` **829**, 830
- action expression in the UML **41**
- action state in the UML **41**
 - symbol **41**
- activity diagram in the UML **41**, 47
- activity in the UML **41**
- ad-hoc constraint (C++20 concepts) **656**
- adapter 543
- add an integer to a pointer 208
- adding strings **38**
- addition 30, 31
 - compound assignment operator, += **57**
- address operator (&) 193, 194, 195, 424
- adjacent_difference algorithm 621
- adjacent_find algorithm 620
- ADL (argument-dependent lookup) 637
- advance function **652**
- aggregate initialization 673
- aggregate type **324**, 328, 670
 - designated initializer 325
- aggregation **308**
- aiming a derived-class pointer at a base-class object 355
- air-traffic-control systems xxi
- alert escape sequence ('\a') 25
- algebraic expression 30
- <algorithm> header 112, **444**, 452, 501, 523, 556
- algorithms (standard library) **507**, 518
 - accumulate **174**, 596, 605
 - binary_search **168**
 - copy_backward **585**
 - for manipulating containers 103
 - for_each **455**
 - gcd 596, **596**
 - iota 596, **597**
 - is_sorted **501**
 - iter_swap 582, **583**
 - lcm 596, **597**
 - max **281**, 594, 594
 - min 594, **594**
 - minmax 594, **595**
 - multimap **515**
 - partial_sum 596, **598**
 - reduce 596, **597**
 - separated from container 558
 - sort **168**, 759
 - specialized memory 621
 - swap 582, **583**
 - swap_ranges 582
- alias 131
 - declaration (`using`) **394**
 - for a type **394**, 680
 - for the name of an object 302
- all
 - algorithm 548
 - range adaptor (C++20) 612
- all_of algorithm 620
 - ranges version (C++20) 578, **580**
- allocate memory 112, **425**, 425, 427
- allocator_type 510
- alphanumeric character **262**
- ambiguity problem 399, 401
- angle brackets (< and >) 137
 - in templates **630**
- anonymous function 176, 560
- any algorithm 548
- <any> header 113
- any_of algorithm **581**, 620
 - ranges version (C++20) 578, **581**
- append **224**
- append data to a file 241
- Apple
 - Xcode xxv, xliii
- arbitrary precision integers
 - BigInteger class **61**
- archive file **94**
- argument coercion **109**
- argument-dependent lookup (ADL) 637
- arguments in correct order 107
- arguments passed to member-object constructors 308
- arithmetic
 - compound assignment operators **57**
 - function object **604**
 - operator xxvi, **30**
 - overflow **492**
 - underflow **492**
- “arity” of an operator 424

- array
 - built-in 199
 - C style 190
 - pointer based 190
- array class template **154**, 482, 508
 - bounds checking 156, 157, **157**
 - container xxvi
 - multidimensional **170**
- <array> header 111, 155, **169**
 - to_array function (C++20) **191**, 201
- array names decay to pointers **199**
- array subscript operator ([]) 436
- arrow member selection operator (->) 291, 292, 316
- as_const function (C++17) **676**
- ASCII (American Standard Code for Information Interchange) character set 85, 216
- assert
 - contract keyword **497**
 - macro **483**, 495
 - macro to disable assertions 483
- assertion **483**
- assign
 - addresses of base-class and derived-class objects to base-class and derived-class pointers 352
 - class objects 305
 - one iterator to another 517
- assign member function
 - list **530**
 - string **224**
- assignment operator **57**, 304, 424, 513
 - *= 57
 - /= 57
- assignment operator (cont.)
 - %= 57
 - += **57**
 - = 57
 - = **29**
 - default **304**
- assignment statement 29
- associative container 516, 533, 535
 - insert function 534, 538
 - map 533
 - multimap 533
 - multiset 533
 - ordered 508, **509**, 533
 - set 533
 - unordered 508, 509, 511, 533
 - unordered_map 533
 - unordered_multimap 533
 - unordered_multiset 533
 - unordered_set 533
- associative container member functions
 - contains **535**
 - count **534**, 540
 - equal_range **536**
 - extract member function (C++17) 512
 - find **535**
 - insert **536**, 540
 - lower_bound **536**
 - merge member function (C++17) 533
 - upper_bound **536**
- associativity of operators **31**
- asterisk (*) **30**
- asynchronous
 - concurrent threads 782
 - event **481**
 - programming 103
 - task 836, 849
 - task completes 834
- at member function 524
 - array **157**, 495
 - string **224**, 422
 - vector **184**, 495
- <atomic> header **816**
- atomic operation **784**
- atomic pointer 819
- atomic types **816**
 - std::atomic class template **817**
 - std::atomic_ref class template (C++20) **817**, 820
 - thread safety 757
- atomic_ref class template (C++20) **817**, 820
- attribute
 - [[fallthrough]] **83**
 - in the UML 19
 - of a class 18
 - of an object 19
- audit
 - contract level **498**
 - level precondition 502
- auto keyword **172**, 521
- automatically destroyed 126
- average calculation 48, 49, 50
- avoid
 - naming conflicts 315
 - protected data 406
 - repeating code 297
- await_ready function of an awaitable object (coroutines) **855**
- await_resume function of an awaitable object (coroutines) **855**
- await_suspend function of an awaitable object (coroutines) **855**
- awaitable object (coroutines) **849**, 855
 - await_ready function **855**
 - await_resume function **855**
 - await_suspend function **855**
- axiom contract level **498**

B

- back member function
 - queue **545**
 - sequence containers **524**
 - span class template (C++20) **214**
 - vector **257**
- back_inserter function template **571**, 589
- background_executor (conurrencpp) **845**
- backslash
 - \ 25
 - escape sequence, \\ 25
- bad data **253**, 254
- bad_alloc exception **425**, **487**, 491
- bad_cast exception **491**
- bad_typeid exception **491**
- Bancila, Marius
 - blog xxxv
- banking systems xxi
- bar chart 164
 - printing program 164
- bar of asterisks 164
- barrier (C++20) 820, **823**
- <barrier> header (C++20) 113, **823**
- base case(s) **140**, 144, 146
- base class **336**, 341
 - catch 492
 - default constructor 350
 - destructor (protected and non-virtual) 377
 - destructor (public and virtual) 377
 - exception 491
 - initializer **345**
 - pointer to a base-class object 352
 - pointer to a derived-class object 352
 - private member 405
 - subobject **402**
- base *e* 104
- base-10 number system 104
- basic exception safety guarantee **476**
- basic searching and sorting algorithms of the standard library 578
- basic_ios class 402
- basic_iostream class 402
- basic_istream class 402
- basic_ostream class 402
- begin
 - function **169**, 200
 - header <array> **169**
 - member function of containers 512
 - member function of first-class containers **513**
- beginning
 - of a file 244
 - of a stream 245
- behavior
 - of a class 18
- bell 25
- bidirectional iterator 515, 516, 526, 533, 537, 539, 664
 - operations 517
- bidirectional_iterator concept (C++20) 559, 587
- bidirectional_range concept (C++20) 559, 585, 587, 588, 589
- big data 222, 250
- Big Four C++20 features 628
- Big O notation **549**, 551
- BigInteger class **61**
 - pow member function **64**
- binary function 605
 - object **605**
- binary left fold **685**, 686, 689
- binary operator **29**, 30, 90
- binary predicate function 528, 567, 580, 586, 589, 592
- binary right fold **686**, 689
- binary search 500
 - algorithm 551
- binary search (cont.)
 - binary_search standard library algorithm **168**, 170, 620
 - binary_search standard library algorithm ranges version (C++20) **580**
- binary tree **508**
- binary_semaphore (C++20) **827**
- <bit> header 113
- bit manipulation 547
- Bitcoin 809
- bitset 509, 547
- <bitset> header 111
- bitwise
 - assignment operators 548
 - left-shift operator (<<) 416
 - operators xxxi
 - right-shift operator (>>) 416
- block **34**, 46, 124, 125
 - of memory 531
 - scope **124**
 - thread until a lock is released **787**
- blockchain 809
- blocked thread state **769**
- blogs
 - Bancila, Marius xxxv
 - Boccaro, Jonathan xxxv
 - Filipek, Bartlomiej xxxv
 - Grimm, Rainer xxxv
 - Microsoft's C++ Team xxxv
 - O'Dwyer, Arthur xxxv
 - Sutter's Mill xxxv
- Boccaro, Jonathan
 - blog xxxv
- body
 - function **24**
 - if statement 32
- Bohm, C. 40
- bool
 - contextual conversion **439**
 - data type **44**
- boolalpha stream manipulator **37**

- Boolean 44
 - Boolean values in JSON 326
 - Boost C++ libraries xxiv
 - Boost.Log logging library 494
 - Boost.Multiprecision library (precise floating-point calculations) 75
 - born thread state **768**
 - bounds checking **157**
 - braced 443
 - braced initializer **27**, 426
 - list 443
 - list as constructor argument 443
 - list for custom classes 443
 - narrowing conversion 109
 - braces ({}) 24, 34, 46
 - not required 83
 - break statement **83**, 86
 - brittle
 - base-class problem 406
 - software **406**
 - broadcast operations 440
 - buffer **777**
 - buffer overflow 158
 - build level (contracts) **502**
 - built-in array xxvii, 190, 199
- C**
- C xxxiv
 - C-like pointer-based array 509
 - C-string xxvii, 190, **216**
 - C-style arrays 190
 - C-style string 190
 - C++
 - code repositories xxxiv
 - Language Reference xxxv
 - open-source community xxxiv
 - C++ Core Guidelines xxiii, xxxi, 746
 - explicit single-parameter constructor **277**
 - Guidelines Support Library 110
 - override 361
 - C++ documentation xxxiv
 - C++ *How to Program, Eleventh Edition* xxxvii
 - C++ language documentation (Microsoft) xxxv
 - C++ preprocessor 23
 - C++ standard library xxiv, 22, 103
 - array class template **154**
 - container **154**
 - exception types 491
 - headers 111
 - string class **35**, 273
 - <string> header **37**
 - vector class template 181
 - C++ Standards Committee xxxv
 - C++11 xxi
 - auto keyword **172**
 - braced initialization **27**
 - braced initializers as constructor arguments 443
 - end container member function **522**
 - crbegin container member function **522**
 - crend container member function **522**
 - <cstdlibint> header 207
 - default special member function **361**, 444
 - default type arguments for function template type parameters 678
 - delegating constructor **298**
 - fixed-size integer types 207
 - in-class initializer **285**
 - launch enum **814**
 - list initialization 541
 - noexcept **448**
 - nullptr constant **192**
 - override 358, **361**
 - <random> header **113**
 - <regex> header **261**, 265
 - C++11 (cont.)
 - scoped enumeration (enum class) **120**
 - shrink_to_fit container member function for vector and deque **522**
 - specifying an enum's integral type 123
 - static_assert declaration **659**
 - std::async function template 808, **814**
 - std::begin function 200
 - std::call_once **816**
 - std::condition_variable class **787**
 - std::condition_variable_any class **805**
 - std::end function 200
 - std::forward_list class template 508
 - std::future class template 814
 - std::iota algorithm 621
 - std::lock_guard class **791**
 - std::minmax algorithm 594, **595**
 - std::move function **438**, 439
 - std::mutex class **787**, 788
 - std::once_flag **816**
 - std::packaged_task function template **815**
 - std::promise **814**
 - std::random_device random-number source **118**, 123
 - std::shared_future class template 815
 - std::shared_lock class **804**
 - std::shared_mutex class **804**
 - std::shared_ptr class template 428

C++11 (cont.)

std::this_thread::
 get_id function **772**
 std::this_thread::
 sleep_for function
 773
 std::this_thread::
 sleep_until function
 773
 std::thread **771**
 std::to_string func-
 tion **235**
 std::unique_lock class
 788, 789
 std::unique_ptr class
 template **428**, 430
 std::unordered_mu-
 ltimap class template
 509
 std::unordered_mu-
 ltiset class template
 509
 std::unordered_set
 class template 509
 std::weak_ptr class
 template 428
 stod function 235
 stof function 235
 stoi function 235
 stold function 235
 stoll function 235
 stoul function 235
 stoull function 235
 thread_local storage
 class **758**
 <tuple> header 111
 variadic template **679**

C++14 xxi

digit separator ' **63**
 generic lambdas **176**
 heterogeneous lookup (as-
 sociative containers)
 537
 std::make_unique func-
 tion template **428**, 430

C++14 (cont.)

std::quoted stream ma-
 nipulator **246**
 string-object literal **420**
 variable template **678**

C++17 xxi

<chrono> header **761**
 class template argument
 deduction (CTAD) **158**
 constexpr if **699**
 contiguous iterator 515
 <execution> header **762**
 execution policy **762**, 763
 extract member func-
 tion of associative con-
 tainers 512
 [[fallthrough]] attri-
 bute **83**
 <filesystem> header
 241
 fold expression 628, 682
 merge member function
 of associative containers
 533
 std::as_const function
 676
 std::exclusive_scan
 parallel algorithm **766**
 std::execution::par
 execution policy **762**,
 763
 std::execution::
 par_unseq execution
 policy **763**
 std::execution::par-
 allel_policy class
 763
 std::execution::par-
 allel_sequenced_
 policy class **763**
 std::execution::seq
 execution policy **763**
 std::execution::se-
 quenced_policy class
 763

C++17 (cont.)

std::execution::un-
 seq execution policy
 763
 std::execution::un-
 sequenced_policy
 class **763**
 std::filesystem-
 tem::path **241**
 std::for_each_n paral-
 lel algorithm **766**
 std::inclusive_scan
 parallel algorithm **766**
 std::optional class
 template 191
 std::reduce parallel al-
 gorithm **766**
 std::scoped_lock class
 791
 std::string_view 190,
 236, 274
 std::transform_ex-
 clusive_scan parallel
 algorithm **766**
 std::transform_in-
 clusive_scan parallel
 algorithm **766**
 std::trans-
 form_reduce parallel
 algorithm **766**
 <string_view> header
 236
 structured binding 595
 unpack elements via struc-
 tured binding **577**

C++20 xxi

abbreviated function tem-
 plate 634, **634**
 ad-hoc constraint in con-
 cepts **656**
 <barrier> header **823**
 bidirectional_itera-
 tor concept 587
 bidirectional_range
 concept 585, 587, 588,
 589
 “big four” features 628

C++20 (cont.)

- C++ standard document xxxv
- co_await operator (coroutines) 834
- co_return statement (coroutines) 834, **848**
- co_yield expression (coroutines) 834, **837**, 839
- <compare> header 460
- concept keyword **648**
- concepts **411**, 556, 558, 636, 640, 652
- concepts by header **642**
- <concepts> header **641**
- conjunction in a constraint or concept **642**
- constexpr function **699**
- constrained auto **650**
- constraint expression in a concept **640**, 648
- constraint in concepts **640**, 641
- contiguous_iterator concept 564
- contracts (pushed to a later standard) 496
- coroutine 834
- disjunction in a constraint or concept **642**
- ends_with member function of class string **38**
- forward_iterator concept 569, 576
- forward_range concept 569, 571, 576, 580, 586, 593
- indirectly_copyable concept 561
- indirectly_readable concept 561
- indirectly_swappable concept 583
- indirectly_writable concept 561, 572

C++20 (cont.)

- input_iterator concept 570, 587
- input_or_output_iterator concept 565
- input_range concept 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595
- iterator concepts 559
- <latch> header **820**
- output_iterator concept 564
- output_range concept 564
- permutable concept 575
- projection in a ranges algorithm 567
- projection in std::ranges algorithms 608
- random_access_iterator concept 575, 579, 600
- random_access_range concept 575, 579
- range **177**, 507
- range adaptor **611**
- range concepts 559
- <ranges> header **177**
- ranges library **177**, 253
- requires clause **640**
- requires expression **654**
- <semaphore> header **827**
- sentinel of a range 525
- standard concepts by header **642**
- std::all_of algorithm (ranges) 578, **580**
- std::any_of algorithm (ranges) 578, **581**
- std::atomic_ref class template **817**, 820
- std::barrier 820, **823**

C++20 (cont.)

- std::binary_search algorithm (ranges) 578, **580**
- std::binary_semaphore **827**
- std::copy algorithm (ranges) **525**, 560
- std::copy_backward algorithm (ranges) 584, **585**
- std::copy_if algorithm (ranges) 584, **587**
- std::copy_n algorithm (ranges) 584, **587**
- std::count algorithm (ranges) 574, **575**, 577
- std::count_if algorithm (ranges) 574
- std::counting_semaphore **827**
- std::equal algorithm (ranges) 566, **566**
- std::equal_range algorithm (ranges) 592
- std::fill algorithm (ranges) **563**, 564
- std::fill_n algorithm (ranges) **563**, 564
- std::find algorithm (ranges) **578**
- std::find_if algorithm (ranges) 578, **579**
- std::find_if_not algorithm (ranges) 578, **582**
- std::for_each algorithm (ranges) **561**
- std::format function from header <format> **65**, 98
- std::generate algorithm (ranges) **563**, 564
- std::generate_n algorithm (ranges) **563**, 565, 565
- std::includes algorithm (ranges) 589, **590**

C++20 (cont.)

std::inplace_merge algorithm (ranges) **588**
 std::jthread **771**, **776**
 std::latch **820**, **820**, **821**
 std::lexicographical_compare algorithm (ranges) **566**, **568**
 std::lower_bound algorithm (ranges) **592**, **593**
 std::make_heap algorithm (ranges) **600**
 std::max_element algorithm (ranges) **574**, **576**
 std::merge algorithm (ranges) **584**, **586**
 std::min_element algorithm (ranges) **574**, **576**
 std::minmax algorithm (ranges) **595**
 std::minmax_element algorithm (ranges) **574**, **576**
 std::mismatch algorithm (ranges) **566**, **567**
 std::move algorithm (ranges) **586**
 std::move_backward algorithm (ranges) **586**
 std::none_of algorithm (ranges) **578**, **581**
 std::pop_heap algorithm (ranges) **602**
 std::push_heap algorithm (ranges) **601**
 std::ranges namespace **525**, **560**, **561**, **563**, **566**, **568**, **572**, **574**, **578**, **582**, **584**, **588**, **589**, **592**, **594**, **599**
 std::remove algorithm (ranges) **568**, **569**
 std::remove_copy algorithm (ranges) **568**, **570**

C++20 (cont.)

std::remove_copy_if algorithm (ranges) **568**, **572**
 std::remove_if algorithm (ranges) **568**, **571**
 std::replace algorithm (ranges) **572**, **572**
 std::replace_copy algorithm (ranges) **572**, **573**
 std::replace_copy_if algorithm (ranges) **572**, **574**
 std::replace_if algorithm (ranges) **572**, **573**
 std::reverse algorithm (ranges) **584**, **587**
 std::reverse_copy algorithm (ranges) **588**, **589**
 std::same_as concept **649**
 std::set_difference algorithm (ranges) **589**, **591**
 std::set_intersection algorithm (ranges) **589**, **591**
 std::set_symmetric_difference algorithm (ranges) **589**, **591**
 std::set_union algorithm (ranges) **592**
 std::shuffle algorithm (ranges) **574**, **575**
 std::sort algorithm (ranges) **578**, **579**, **609**
 std::sort_heap algorithm (ranges) **601**
 std::span class template of header **191**, **210**
 std::starts_with member function of class string **38**

C++20 (cont.)

std::stop_callback for cooperative cancellation **808**
 std::stop_source for cooperative cancellation **807**
 std::stop_token for cooperative cancellation **807**
 std::swap_ranges algorithm (ranges) **583**, **584**
 std::to_array function of header <array> **191**, **201**
 std::transform algorithm (ranges) **574**
 std::unique algorithm (ranges) **584**, **586**
 std::unique_copy algorithm (ranges) **588**, **589**
 std::upper_bound algorithm (ranges) **592**, **593**
 <stop_token> header **805**
 templated lambda **636**
 three-way comparison operator (<=>) **460**, **511**
 view **177**, **507**, **611**
 viewable_range **611**
 weakly_incrementable concept **561**
C++20 for Programmers code download **xliii**
 C++20 Fundamentals LiveLessons videos **xxxvii**
 C++20 modules **xxiii**
 transition from the preprocessor **712**
 C++20 ranges
 | operator in a range pipeline **178**
 pipeline **178**
 std::views::filter **178**, **179**
 std::views::iota **178**

- C++23 xxi, 411
 - concurrent map 831
 - concurrent queue 830
 - contracts (could be later than C++23) 496
 - modular standard library 746
 - ranges enhancements 622
 - `std::mdarray` container 173
- C++26 411
- Caesar cipher 148
- calculations 41
- callback function **834**
- calling functions by reference 195
- camel case **28**
- capacity
 - of a string **227**
 - of a vector **519**
- capacity member function
 - of string **228**
 - of vector **519**
- capturing variables in a lambda **257**, 456
- caret (^) regex metacharacter **263**
- carriage return ('\r') escape sequence 25
- cascading
 - member function calls **316**, 317, 319
 - stream insertion operations **30**
- case insensitive 266
 - regular expression 261
- case keyword **83**
- case sensitive **28**, 266
 - regular expression 261
- case studies xxv
- casino 119
- `<cassert>` header 112, **483**
- cast operator **52**, 210, 463
 - cast away `const`-ness 670
 - overloaded **454**
- catch block **185**
- catch exceptions in constructors 484
- catch handler 476, 480
 - all exceptions with `catch(...)` 492, 493
 - base-class exception 492
- catch related errors 492
- `catch(...)` (catch all exceptions) 492, 493
- `cbegin`
 - member function of containers 512
 - member function of vector **521**
- `<cctype>` header 112
- `ceil` function 104
- `ceil`
 - member function of containers 512
 - member function of vector **522**
- `cereal` header-only library 251, **327**
 - `JSONInputArchive` 331
 - `JSONOutputArchive` 329
- `cerr` (standard error stream) 239
- `<cfloat>` header 112
- chain of constructor calls **349**
- chain of destructor calls **350**
- chaining stream insertion operations **30**
- char data type 28, 110
- character array 216
- character class (regular expressions) **261**, 262
- character constant **216**
- character literal 85, **85**
- character presentation 112
- character sequence 246, 274
- character string **24**
- `<chrono>` header 111, 284, **761**
 - `duration_cast` **761**
 - `steady_clock` **761**
- `cin` (standard input stream) 29, 239, 242
- cipher
 - Caesar 148
 - substitution **148**
 - Vigenère 148, 149, 150
- ciphertext **148**
- circular buffer **795**
- circular wait (necessary condition for deadlock) 770
- `clamp` algorithm 621
- Clang C++ xxiii, xliii, 4
 - `clang++` in a Docker container 4
- `clang-tidy` static analysis tools xxxii, xlvi
- class **18**
 - `class` keyword 137, 273, **273**, 630
 - constructor **275**
 - data member **19**
 - default constructor **278**
 - development 430
 - diagram in the UML **340**
 - hierarchy **339**, 362
 - implementation programmer **290**
 - interface **284**
 - interface described by function prototypes **107**
 - invariant **295**, 495
 - `public` services **284**
- class-average problem 48, 51
- class scope **124**, 287, 291
 - `static` class member 320
- class template **155**, 409, 627, 629
 - definition 629
 - member-function templates 631
 - scope 634
 - specialization 629, 630
 - Stack 630, 632

- class template argument deduction (CTAD) **158**, 181, 523, 537, 568, 673, 676
- class template specialization **155**
- classes
 - array class template **154**
 - bitset 509, **547**
 - deque 518, **531**
 - exception **491**
 - forward_list 518
 - invalid_argument 492
 - list 518, **526**
 - multimap 539
 - MyArray 432
 - numeric_limits 63
 - out_of_range exception class 185
 - priority_queue **546**, 599, 600, 601
 - queue **545**
 - runtime_error **472**, 480
 - set 537
 - stack **543**
 - steady_clock **761**
 - string **35**, 273
 - system_clock **761**
 - tuple **679**
 - unique_ptr **428**
 - vector 180
- cleaning data 260
- clear member function of containers 512, 526
- client
 - code 351
 - of a class **279**
- client-code programmer **290**
- <climits> header 112
- clog (standard error buffered) 239
- close member function of ofstream **243**
- closed set of types **391**
- cloud 326
- cloud-based services **326**
- cmath **265**
- <cmath> header 77, 103, 111
 - isnan function **256**
 - list of functions 104, 105
 - mathematical special functions **105**
- co_await expression (C++20) **849**
- co_await operator (C++20) 834
- co_return statement (C++20) 834, **848**
- co_yield expression (C++20) 834, **837**, 839
- code **19**
- code download xliii
- code repositories xxxiv
- Coffman, E. G. 770
- coin tossing 114
- collision in a hashtable **552**
- colon (:) 399
 - in inheritance **345**
- column **170**
- column headings 157
- combining control statements
 - in two ways 92
- comma (,) 75
- comma operator (,) **75**, 145, 690
- comma-separated list 27, 34, 75
 - of base classes 399
 - of parameters 107
- command-line argument **217**
- Command Prompt window 6
- comment **23**, 28
 - multiline **23**
 - single-line **23**
- CommissionEmployee class
 - header 369
 - implementation file 369
 - test program 343
- common programming errors xxiii
- common range 524, 557, 560
- common range adaptor (C++20) 612
- communications systems xxi
- CommunityMember class hierarchy 339
- commutative operators **459**
- comparator function object 533, 539
 - less **533**, 546
- <compare> header 113, 460
- compare iterators 517
- compare member function of class string **226**
- comparing strings 225
- compilation error 57
- compile 679
- compile a header as a header unit 714
- compile time
 - calculations 628
- compile-time
 - constant 679
 - polymorphism **408**, 410, 513, **628**, 629
 - predicate 640
 - programs that write code 628
 - recursion 682, 683
 - static polymorphism **628**
- compiler 23, 53
 - Apple Xcode xliii
 - Clang C++ xxiii
 - g++ 11
 - GNU C++ xxiii, xliii
 - GNU g++ 4
 - Microsoft Visual Studio xliii
 - Visual C++ xxii
 - Visual Studio Community edition 4
 - Xcode on macOS 4
- Compiler Explorer xxxix
 - website (godbolt.org) **498**
- pthread compiler flag 771
- compiler warnings
 - enable 93
- completion function **823**, 826
 - barriers **823**

- component 18
- composable 177
- composable views 177, 611
- composition 308, 311, 337, 341
- compound assignment operators 57, 59
- compound interest 75
- compound requirement in C++20 concepts 654, 655
 - > 656
- compound statement 34
- compression
 - run-length encoding 94
- computing the sum of the elements of an array 163, 174
- concatenate 224
 - stream insertion operations 30
- concept-based overloading (C++20) 411, 652, 659, 693, 699
 - concept overloading 411
- concept keyword (C++20) 648
- concepts (C++20) 411, 558, 636, 640, 652
 - > in a compound requirement 656
 - ad-hoc constraint 656
 - bidirectional_iterator 559, 587
 - bidirectional_range 559, 585, 587, 588, 589
 - compound requirement 654, 655
 - concept keyword 648
 - conjunction 642
 - constraint 640, 641
 - constraint expression 640, 648
 - contiguous_iterator 559, 564
 - contiguous_range 559
 - custom 648
- concepts (C++20) (cont.)
 - disjunction 642
 - forward_iterator 559, 569, 576
 - forward_range 559, 569, 571, 576, 580, 586, 593
 - indirectly_copyable 561
 - indirectly_readable 561
 - indirectly_swappable 583
 - indirectly_writable 561, 572
 - input_iterator 559, 570, 587, 653
 - input_or_output_iterator 565
 - input_range 559, 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595
 - iterators 559
 - listed by header 642
 - logical AND (&&) operator
 - in a constraint 642
 - logical OR (||) operator
 - in a constraint 642
 - nested requirement 654, 656
 - output_iterator 559, 564
 - output_range 559, 564
 - permutable 575
 - random_access_iterator 559, 569, 570, 575, 579, 600, 653
 - random_access_range 559, 575, 579, 600, 601, 602
 - ranges 559
 - requires clause 640
 - requires expression 654
- concepts (C++20) (cont.)
 - simple requirement 654, 654
 - standard 640
 - std::floating_point 641, 648
 - std::integral 641, 648
 - std::same_as 649
 - type requirement 654, 655
 - weakly_incrementable 561
- <concepts> header (C++20) 113, 641
- concrete class 362
- concrete derived class 365
- concurrency support library 836
 - background_executor 845
 - executor 836
 - inline_executor 845, 845
 - install 837
 - result 841
 - runtime 841, 843
 - submit function of an executor 844
 - task 836, 841
 - thread_executor 844
 - thread_pool_executor 841, 844, 844
 - timer 836
 - utility functions 836
 - when_all function 848
 - when_any function 849
 - worker_thread_executor 845
- concurrent container
 - Google Concurrency Library (GCL) 830
 - Microsoft Parallel Patterns Library 830
- concurrent map (C++23) 831
 - reference implementation 831
- concurrent operations 756

- concurrent programming
 - 103, **757**
 - with a simple sequential-like coding style 834
- concurrent queue (C++23) 830
 - reference implementations 830
- concurrent threads 783
- condition **31**, 47, 79
 - Yoda 93
- condition variable 789
- condition_variable
 - wait function **789**
- condition_variable class **787**
- <condition_variable>
 - header (C++11) 112, **787**
- condition_variable_any class **805**
- conditional expression **47**
- conditional operator, ?: 47
- confusing equality (==) and assignment (=) operators 92
- conjunction in a C++20 constraint or concept **642**
- const 306
 - keyword 115
 - member function **274**, 306
 - member function on a non-const object 307
 - objects and member functions 307
 - qualifier 162
 - qualifier before type specifier in parameter declaration 131
 - version of operator[] 453
- const_cast
 - cast away const-ness **670**
- const_iterator 510, 512, 516, 535
- const_pointer 510
- const_reference 510
- const_reverse_iterator 510, 512, 516, 522
- constant
 - compile-time 679
- constant integral expression 85
- constant pointer
 - to an integer constant 204
 - to constant data 202, 204, 205
 - to nonconstant data 202, **204**
- constant running time **550**
- constant variable 162
- constexpr function (C++20) **699**
- constexpr function **699**
- constexpr if (C++17) **699**
- constexpr qualifier **162**, 162
- constrained auto (C++20) **650**
- constraint **640**, 648
- constraint (C++20 concepts) **640**, 641
- constraint expression (C++20 concepts) **640**, 648
- constructor **275**, 278
 - braced-initializer list **443**
 - call chain 349
 - conversion **462**, 464
 - copy 446
 - default arguments 296
 - exception handling 483
 - explicit 464
 - function prototype 285
 - in a class hierarchy 349
 - injection **386**
 - multiple parameters 280
 - single argument 464, 465
- constructors and destructors called automatically 298
- consumer 757, **776**
 - thread **777**
- container 103, 111, 436, **506**, 508
 - begin function 512
- container (cont.)
 - cbegin function 512
 - cend function 512
 - clear function 512
 - crbegin function 512
 - crend function 512
 - empty function 512
 - end function 512
 - erase function 512
 - insert function 513
 - map associative container 533
 - map class template 509
 - max_size function 513
 - multimap associative container 533
 - multimap class template 509
 - multiset associative container 533
 - multiset class template 509
 - nested type names 672
 - priority_queue class template 509
 - queue class template 509
 - rbegin function 512
 - rend function 512
 - sequence **508**
 - set associative container 533
 - set class template 509
 - size function 513
 - special member functions 511
 - stack class template 509
 - swap function 513
 - unordered_map associative container 533
 - unordered_multimap associative container 533
 - unordered_multiset associative container 533
 - unordered_set associative container 533
- container (Docker) xxxiv, **xlv**

- container adaptor 508, **509**, 509, 516, 543, 543
 - priority_queue **546**, 599, 600, 601
 - queue **545**
 - stack **543**
- container adaptor functions
 - pop **543**
 - push **543**
- container in the C++ standard library **154**
- container member function complete list 510
- contains function of associative container **535**
- contextual conversion 454, 466
- contextual conversion to bool **439**
- contiguous iterator (C++17) 515, 558
- contiguous_iterator concept (C++20) 559, 564
- contiguous_range concept (C++20) 559
- continuation mode (for contract violations) **500**
- continue statement **86**, 87
- contract 495, 496
 - assert contract keyword **497**
 - attributes 497
 - audit contract level **498**
 - axiom contract level **498**
 - build level **502**
 - continuation mode **500**
 - contract_violation 502
 - default contract level **498**
 - default violation handler **500**
 - design by contract **496**
 - disable contract checking 500
 - early access implementations (GNU C++) 498
 - contract (cont.)
 - ensures contract keyword **497**, 498
 - expects contract keyword **497**, 498
 - experimental implementation 471
 - handle_contract_violation default contract violation handler **500**
 - level 498, 502, 503
 - post contract keyword (GNU C++ early access implementation) **499**
 - pre contract keyword (GNU C++ early access implementation) **499**
 - proposal **497**
 - violation **500**, 502
 - violation handler **503**
 - contract_violation object 502
 - control statement xxvi, 41, 43
 - do...while 78, 79
 - for 42, **71**, 72, 75, 77
 - if **31**
 - nesting **43**
 - stacking **43**
 - switch **80**
 - while 47, 70
 - control variable **70**, 71, 72
 - controlling expression of a switch **83**
 - converge on the base case 146
 - conversion, contextual conversion to bool **439**
 - conversion constructor 418, **462**, 464
 - conversion operator 418, **454**, 463
 - explicit 465
 - convert among user-defined types and built-in types 463
 - convert between types 462
 - convert lowercase letters 112
 - convert strings to floating-point types 235
 - convert strings to integral types 235
 - cooperative **805**
 - cooperative cancellation 776, **805**, 807
 - std::stop_callback **808**
 - std::stop_source **807**
 - std::stop_token **807**
 - cooperative multitasking 835
 - cooperative thread cancellation 805
 - coordination types (thread synchronization) 820
 - copy 472
 - copy algorithm 441, **444**, 523, 620
 - ranges version (C++20) **525**, 560
 - copy-and-swap idiom 447, 459
 - strong exception guarantee 477
 - copy assignment operator (=) xxviii, 278, 417, **431**, 446, 513
 - overloaded **420**
 - copy-constructible type 472
 - copy constructor xxviii, 278, **306**, 311, 417, 421, 431, 434, 437, 445, 446, 511, 513
 - default 311
 - copy of the argument 202
 - copy semantics xxviii, 417, **432**
 - copy_backward algorithm **585**, 620
 - ranges version (C++20) 584, **585**
 - copy_if algorithm 620
 - ranges version (C++20) 584, **587**
 - copy_n algorithm 620
 - ranges version (C++20) 584, **587**
 - CopyConstructible 513

- coroutine **840**
 - coroutine (C++20) 834
 - awaitable object **855**
 - co_yield expression **837**, 839
 - coroutine frame **855**
 - coroutine state **855**
 - coroutine support library 835, 849
 - coroutine_handle **855**
 - generator **837**
 - generator coroutine support library (Sy Brand) 837
 - promise object **854**
 - stackless **840**
 - suspend_always **854**
 - suspend_never **854**
 - suspension point **855**
 - <coroutine> header 113
 - <coroutine> header (C++20) **854**
 - coroutine libraries
 - conurrencpp **836**
 - cppcoro **836**
 - folly::coro **836**
 - generator (Sy Brand) **836**, 837
 - correct number of arguments 107
 - correct order of arguments 107
 - cos function 104
 - cosine 104
 - count algorithm 620
 - ranges version (C++20) 574, **575**, 577
 - count function
 - of multimap 540
 - count function of associative container **534**
 - count_down member function of a std::latch **821**
 - count_if algorithm 620
 - ranges version (C++20) 574
 - count_if ranges algorithm (C++20) **257**, 258
 - counted range adaptor (C++20) 612
 - counter **48**
 - counter-controlled iteration xxvi, **48**, 48, 52, 70, 71, 146
 - counting loop 71
 - counting_semaphore (C++20) **827**
 - cout (standard output stream) 24, 26, 239
 - .cpp filename extension 719
 - cppcheck static analysis tools xxxii, xlvi
 - cppcoro coroutines library **836**
 - cpplang Slack channel xlvi
 - .cppm filename extension 719
 - crafting valuable classes with operator overloading 430
 - craps simulation 119, 120
 - crbegin
 - member function of containers 512
 - member function of vector **522**
 - Create a New Project** dialog in Visual Studio Community Edition 5
 - create a sequential file 240
 - create an array object from a built-in array or an initializer list 201
 - create an object (instance) 36, 271
 - create your own data types 30
 - CreateAndDestroy class
 - definition 299
 - member-function definitions 300
 - crend
 - member function of containers 512
 - member function of vector **522**
 - critical section **784**, 787, 788, 795, 827
 - critical sections 788
 - cryptocurrency 809
 - <csdint> header (C++11) 60, 207
 - <cstdio> header 112
 - <stdlib> header 111, 489, 490
 - <cstring> header 112
 - CSV (comma-separated values)
 - .csv file extension **250**
 - file format 222, **250**
 - rapidcsv header-only library 251
 - CTAD (class template argument deduction) **158**, 523
 - <ctime> header 111
 - <Ctrl>-d 82, 242
 - <Ctrl> key 82
 - <Ctrl>-z 82, 242
 - curly braces in format string 66
 - current position in a stream 245
 - cursor **25**
 - custom character class **262**
 - custom concept 648
 - custom exception class 472
 - custom functions xxvi
 - customization points for derived classes 377
- ## D
- dangling pointer **445**
 - dangling reference **131**
 - data
 - mutable **757**
 - data analytics 222, 250
 - data compression **94**
 - lossless **94**
 - lossy 94
 - data-interchange format
 - JSON 326
 - data member **19**
 - data persistence **222**

- data race **783**
- data science 222, 250
- data structure **154**, 506
- data types
 - char 110
 - float 110
 - int **27**
 - long double 110
 - long int 110
 - long long 110
 - long long int 110
 - unsigned char 110
 - unsigned int 110
 - unsigned long 110
 - unsigned long int 110
 - unsigned long long 110
 - unsigned long long int 110
 - unsigned short 110
 - unsigned short int 110
- database 804
- dataset 222, 250
 - Titanic* disaster 253
- date and time utilities 761
- Date class 308
- dates 103
- DbC (design by contract) **496**
- deadlock **769**
 - four necessary conditions 770
 - prevention (Havender) 770
 - process or thread 769
 - sufficient conditions 770
- deallocate memory **425**, 427
- Debug** area (Xcode) 9
- decay to a pointer (array names) **199**
- decimal point 53, 54
- decision 44
 - making xxvi
 - symbol in the UML **44**
- declaration **27**
- declarative programming **175**
- decrement
 - a pointer 208
 - operator, -- **58**, 454
- deduction guide 674
- deep **445**
- deep copy **445**
- deep learning 222, 250
- default 311
- default arguments **132**, 292
 - with constructors 292
- default assignment operator **304**
- default case in a switch **83**, 84, 117
- default constructor **278**, 285, 292, 313, 511
- default contract level **498**
- default copy constructor 311
- default destructor 298
- default special member function 361, 444
 - autogenerate a virtual destructor **361**, 444
- default type argument 604
 - for a type parameter **678**, 678
- default violation handler (contracts) **500**
- default_random_engine **114**
- #define** preprocessing directive **711**
- definition **71**
- Deitel & Associates, Inc. xlii
 - virtual and on-site corporate training xlii
- Deitel, Dr. Harvey M. xli
- Deitel, Paul J. xli
 - Full-Throttle training courses xxxvii
 - Live Instructor-Led Training xxxvii
- delegating
 - constructor **298**
 - to other functions **632**
- delete **425**, 429
 - placement 425
- delete[] (dynamic array deallocation) 426
- deleting dynamically allocated memory 427
- dependency injection **386**
- dependent condition 90
- deprecated 111
- deque class template 508, 518, **531**, 631, 678
 - push_front function 531
 - shrink_to_fit member function **522**
- <deque> header 111, **531**
- dereference
 - a pointer **193**, 196, 203
 - an iterator 513, 515, 517
 - an iterator positioned outside its container 522
- dereferencing operator (*) **193**
- derive one class from another 308
- derived class **336**, 341
 - catch 492
 - customization point 377
 - pointer to a base-class object 352
 - pointer to a derived-class object 352
- descriptive statistics **256**, 256
- deserialization xxvii
- deserializing data **326**
- design by contract (DbC; Bertrand Meyer) **496**
- design pattern 427
- design process **20**
- designated initializer (aggregates) 325
- destructor xxviii, 279, **298**, 417, **431**, 511
 - called in reverse order of constructors 298
 - in a class hierarchy 350
- destructor in a derived class 350
- destructors called in reverse order 350
- destructors should not throw exceptions 483, 486

- detach a thread 775
- device driver
 - polymorphism in operating systems 363
- devirtualization **362**
- diagnostics that aid program debugging 112
- diamond in the UML **41**
- diamond inheritance (in multiple inheritance) **402**
- dice game 119
- die rolling
 - using an array instead of switch 165
- difference_type 510
 - nested type in an iterator 667
- digit 28
- digit separator ' (C++14) **63**
- Dionne, Louis 409
- direct access elements of a container 508
- direct base class **340**, 340
- directly reference a value **192**
- disable assertions 483
- Discord server #include <C++> xlvii
- disjunction in a C++20 constraint or concept **642**
- disk space 488, 490
- dispatch
 - a thread **768**
- display a line of text 22
- distance algorithm **652**
 - std::ranges 663
- distribution (random-number generation) **114**
- DivideByZeroException 476
- divides function object 604
- division 30, 31
 - by zero 471
 - compound assignment operator, /= 57
- do...while iteration statement 42, **78**
- Docker xxxiv, **xliv**
 - Clang C++ container xxiii
 - Clang container 709
 - clang++ container 4
 - container xxxiv, **xliv**, 709
 - Docker Desktop 13, 14
 - Docker Engine 13, 14
 - GCC Docker container 13
 - GNU C++ container xxiii
 - GNU Compiler Collection (GCC) 13
 - GNU Compiler Collection (GCC) container 4, 13
 - image **xliv**
 - Docker Desktop installer xlvii
 - Docker Hub account xlvii
 - documentation
 - C++ xxxiv
 - dot operator (.) **37**, 291, 292, 316, 358, 429
 - dotted line in the UML **42**
 - double-checked locking 815
 - double data type 28, **50**, 109
 - double dispatch **411**
 - double-ended queue 531
 - double-precision floating-point number **77**
 - double quote 25
 - double-selection statement **42**
 - “doubly initializing” member objects 313
 - doubly linked list 508, 526
 - download examples xxii
 - dreamincode.net/forums/forum/15-c-and-c/ xxxvi
 - driver program **35**
 - drop range adaptor (C++20) 612, **615**
 - drop_while range adaptor (C++20) 612, **615**
 - dual-core processor 17
 - duck typing 338, **397**, 409
 - dummy value **50**
 - duplicate keys 533, 539
 - duration_cast function template **761**
 - dynamic binding **358**, 373, 376
 - dynamic casting **409**
 - dynamic data structure 190
 - dynamic memory allocation 220, 417, **425**, 427, 428, 481, 488
 - array of integers 441
 - dynamic_cast 491
 - dynamically determine function to execute 357, 359
 - Dyno library for type erasure **409**

E

 - Easylogging++ logging library 494
 - ECMAScript regular expressions 260
 - Editor area (Xcode) 8, 9
 - efficiency of
 - binary search 551
 - linear search 550
 - element of an array **155**
 - elements range adaptor (C++20) 612, 617
 - else keyword 45
 - embedded parentheses **31**
 - embedded system xxi, 16, 482
 - emplace member function
 - of queue 545
 - of stack 543
 - emplace_after 512
 - emplace_front 512
 - emplace_hint 512
 - Employee abstract base class 364
 - Employee class 308
 - definition showing composition 310

- Employee class (cont.)
 - definition with a `static` data member to track the number of `Employee` objects in memory 321
 - header 366
 - implementation file 366
 - member-function definitions 310, 322
- empty member function
 - of containers 512
 - of `priority_queue` 546
 - of `queue` 545
 - of sequence container **525**
 - of `stack` 543
 - of `string` 228
- empty member function of `string` **38**, 419
- Empty Project** template 5
- empty statement (a semicolon, ;) 46
- empty `string` 37, 228, 272, 274
- enable compiler warnings 93
- encapsulation **19**, 274, 304
- enclosing scope **479**
- end
 - function **169**, 200
 - function of header `<array>` **169**
 - member function of containers 512, **513**
- end of a stream 245
- “end of data entry” 50
- end-of-file (EOF)
 - indicator **82**, 242
 - key combination 242
 - marker **239**
- `ends_with` member function of class `string` (C++20) **38**
- engine (random-number generation) **114**
- `ensures` contract keyword **497**, 498
- Enter* key 29
- `enum`
 - keyword **123**
 - specifying underlying integral type 123
- `enum class` **120**
- enumeration **120**
 - constant **120**
- `equal` algorithm 441, **452**, 620
 - ranges version (C++20) **566**, **566**
- `equal` to 31
- `equal_range`
 - algorithm 620
 - algorithm, ranges version (C++20) 592
 - function of associative container **536**
- `equal_to` function object 604
- equality operators **31**, 32
 - `!=` 44
 - `==` 431
 - `==` and `!=` 44
- `EqualityComparable` 513
- erase 232
 - algorithm from header `<vector>` 570
 - member function of containers 512
 - member function of first-class containers 525
 - member function of `string` **233**
 - member function of `vector` 569
- erase-remove idiom 569, **569**, 570
- `erase_if` algorithm from header `<vector>` 570
- error detected in a constructor 484
- `error_code` class 494
- escape character **25**
- escape sequence **25**, 26, 249
 - `\'` (single-quote character) 25
 - `\"` (double-quote character) 25
 - `\\` (backslash character) 25
 - `\a` (alert) 25
 - `\n` (newline) 25
 - `\r` (carriage return) 25
 - `\t` (tab) 25
- `eText` (Pearson) xxxvii
- examples (download) xxii
- exception **157**, 185, 185, **469**
 - `bad_alloc` **425**
 - handler **185**
 - handling 180
 - `invalid_argument` 235
 - memory footprint 470
 - `out_of_bounds` 431
 - `out_of_range` **185**, 236
 - parameter 185
 - what member function of an exception object **186**
- exception class **491**
 - what virtual function **475**
- exception guarantee
 - copy-and-swap idiom 477
- exception handling 112, **469**
 - flow of control 470, 503
- `<exception>` header 112, **491**
- exception in a thread 771
- exception parameter 474
- exception safe code 476
- exception safety guarantees
 - basic exception safety guarantee **476**
 - no guarantee **476**
 - no throw exception safety guarantee **477**
 - strong exception safety guarantee **477**
- exception types in the C++ standard library 491

- exceptions
 - `bad_alloc` 487
 - `bad_cast` 491
 - `bad_typeid` 491
 - `filesystem_error` 494
 - `length_error` 492
 - `logic_error` 491
 - `out_of_range` 492
 - `overflow_error` 492
 - `underflow_error` 492
- exchange function (header `<utility>`) 448
- exclusive 776
- exclusive resource 776
- `exclusive_scan`
 - algorithm 621
 - parallel algorithm (C++17) 766
- execution
 - parallel 764
- `<execution>` header 113, 762
- execution policy
 - `std::execution::par` 762
- execution policy (C++17) 762, 763
- execution-time overhead 373
- executor (concurrency `cpp` coroutine support library) 841, 844
 - scheduling tasks 836
- `exit` a function 25
- `exit` function 242, 299, 489
- exit point
 - of a control statement 43
- `EXIT_FAILURE` 242
- `EXIT_FAILURE` constant 490
- `EXIT_SUCCESS` 242
- exiting a `for` statement 86
- `exp` function 104
- `expects` contract keyword 497, 498
- expiring value 438
- explicit
 - conversion 53
 - narrowing conversion 110
- `explicit` keyword 277, 465
 - constructor 464
 - conversion operators 465
- exponential “explosion” of calls 145
- exponential complexity 145
- exponential function 104
- exponentiation 77
- `export` (C++20 modules)
 - a block 716
 - a declaration 716, 716, 719, 719
 - a namespace 717
 - a namespace member 717
- `export import` (C++20 modules) 734
- `export module` (C++20 modules) 718
 - declaration for a module interface partition 733
- expression 44, 53
- extensible 337, 351
 - programming language 270
- external iteration 161
- `extract` member function of associative containers (C++17) 512
- extracting data from text 260
- F**
- `fabs` function 104
- Facebook
 - Folly open-source library 409
- factorial 61, 140, 141
 - with `partial_sum` 599
- factorials 599
- fail fast 483
- `[[fallthrough]]` attribute 83
- `false` 32, 44, 146
- fault-tolerant programs 185, 469
- features in a dataset 253
- Fertig, Andreas xxxviii
- Fibonacci series 143, 145, 836
 - generator coroutine 837
- field width 77, 99
- FIFO (first-in, first-out) 509, 531, 545
- file 222, 244
- file compression
 - ZIP 94
- file open mode 241, 243
 - `ios::app` 241
 - `ios::ate` 241
 - `ios::binary` 241
 - `ios::in` 241, 243
 - `ios::out` 241
 - `ios::trunc` 241
- file-position pointer 244
- file processing 103
- file scope 125, 291, 719
- filename 241, 243
- filename extensions
 - `.cpp` 719
 - `.cppm` 719
 - `.h` 272
 - `.ifc` 718
 - `.ixx` 718, 718
 - `.pcm` 719
- `<filesystem>` header (C++17) 113, 241, 494
- `filesystem_error` 494
- `filesystem_error` class 494
- `filesystem::path`(C++17) 241
- Filipek, Bartłomiej
 - blog xxxv
- `fill` algorithm 620
 - ranges version (C++20) 563, 564
- `fill_n` algorithm 620
 - ranges version (C++20) 563, 564
- filter 611
- `filter` range adaptor (C++20) 612
- filtering in functional-style programming 178, 611

- final
 - class 362
 - member function **361**
- final state in the UML **41**
- final value 71
- final_suspend function of a coroutine promise object **854**
- find algorithm 620
 - ranges version (C++20) 578, **578**
- find function of associative container **535**
- find member function of class string 230, **231**
- find member function of string_view **239**
- find_end algorithm 620
- find_first_not_of member function of class string **232**
- find_first_of
 - algorithm 620
 - member function of class string **231**
- find_if algorithm 620
 - ranges version (C++20) 578, **579**
- find_if_not algorithm 620
 - ranges version (C++20) 578, **582**
- find_last_of member function of class string **231**
- finding strings and characters in a string 230
- first
 - data member of pair **536**
 - member function of span class template (C++20) **215**
- first-class container
 - begin member function **513**
 - clear function 526
 - end member function **513**
 - erase function 525
- first-in, first-out (FIFO) 509, 531
 - data structure **545**
- fixed point
 - format **53**
 - value 77
- fixed-size data structure 199
- fixed-size integer types (C++11) 207
- fixed stream manipulator **53**
- flag value **50**
- float data type **50**, 110
- floating-point arithmetic 416
- floating-point calculations (precise)
 - Boost.Multiprecision monetary library 75
- floating-point division 53
- floating-point literal 75, **77**
 - double by default 77
- floating-point number **50**, 50, 52
 - double data type **50**
 - double precision **77**
 - float data type **50**
 - single precision **77**
- floating_point concept **641**, 648
- floor function 104
- flow of control 52
 - exception handling 470, 503
- flow of control in the if...else statement 45
- flow of control of a virtual function call 375
- fmod function 104
- fmodules-ts compiler flag (g++) **714**
- {fmt} library (C++20 text formatting) **65**, 66, 67
- fold expression (C++17) 628, 682, **685**
 - binary left fold **685**, 686
 - binary right fold **686**
 - unary left fold **686**
 - unary right fold **686**
- fold operation
 - binary left 689
 - binary right 689
 - unary left 688
 - unary right 688
- Folly open-source library
 - from Facebook 409
 - Poly **409**
- folly::coro coroutine support library **836**
- font conventions in this book xxxiii
- for iteration statement 42, **71**, 72, 75, 77
 - header **72**
- for_each algorithm 441, **455**, 620
 - ranges version (C++20) **561**
- for_each_n
 - algorithm 620
 - parallel algorithm (C++17) **766**
- format function from header <format> (C++20) **65**, 98
 - <format> header (C++20) 113
 - format function **65**, 98
- format specifier (C++20 text formatting) 99
- format string **66**, 66
 - placeholder **66**
- formatted input/output 246
- formatted string
 - curly braces in a replacement field 66
- formatted text 246
- formatting strings 65
- forums
 - dreamincode.net/forums/forum/15-c-and-c/ xxxvi
 - groups.google.com/g/comp.lang.c++.xxxvi
 - reddit.com/r/cpp/xxxvi
 - stackoverflow.com xxxvi

- forward iterator 515, 527, 558, 583
 - operations 517
 - forward_iterator concept (C++20) 559, 569, 576
 - forward_list class template 508, 518, 527
 - splice_after member function 528
 - <forward_list> header 111, 527
 - forward_range concept (C++20) 559, 569, 571, 576, 580, 586, 593
 - fragile base-class problem 406
 - fragile software **406**
 - free function **287**, 314, 458
 - free store **425**, 427
 - friend
 - of a base class 405
 - of a derived class 405
 - friend function 313
 - can access private members of class 314
 - friendship granted, not taken 313
 - friendship
 - not symmetric 313
 - not transitive 313
 - front
 - member function of queue **545**
 - member function of sequence containers **524**
 - front member function of span class template (C++20) **214**
 - front member function of vector **257**
 - front_inserter function template **571**
 - fstream 240
 - <fstream> header 112, 240
 - full template specialization **697**
 - Full-Throttle training courses xxxvii
 - function xxvi, 18, **24**
 - anonymous 176
 - call overhead 128
 - constexpr **699**
 - constexpr **699**
 - definition 125
 - free **287**, 458
 - header 107
 - hypot **104**
 - overloading **134**
 - parameter **107**
 - parameter list **107**
 - prototype **107**, 108, 125, 129
 - signature **108**, 135
 - that calls itself 139
 - function call operator () 466
 - function object 533, **533**, 539, 553, 557, 603
 - also called a functor 557, **603**
 - arithmetic **604**
 - binary **605**
 - divides 604
 - equal_to 604
 - greater 604
 - greater_equal 604
 - less 604
 - less_equal 604
 - less<int> **533**
 - less<T> 539, 546
 - logical **604**
 - logical_end 604
 - logical_not 604
 - logical_or 604
 - minus 604
 - modulus 604
 - multiplies 604
 - negate 604
 - not_equal_to 604
 - plus 604
 - predefined in the STL 604
 - relational **604**
 - function overloading 182
 - function parameter scope **124**
 - function pointer 220, 374, 557, 605
 - function prototype **107**, 313
 - in a class definition 285
 - function scope **124**
 - function template **137**, 627, 652
 - abbreviated (C++20) 137, 634, **634**
 - unconstrained 637
 - function template specialization **137**, 138, 139
 - function try block **484**, 485, 486
 - <functional> header 112, **604**
 - functional programming 698
 - functional structure of a program 24
 - functional-style programming xxv, 113, 178, 441, 557, 611
 - filtering **178**
 - mapping **179**
 - reduction **163**, 174, 175
 - functor (function object) 557, **603**
 - fundamental type xxvi, **28**, 60
 - bool **44**
 - char 28, 110
 - double **50**
 - float **50**
 - int 57
 - long **60**, 60
 - long double **50**, 77
 - long long 60, 61
 - promotion 53
 - future class template
 - get member function 815
 - <future> header 112, **813**
- ## G
- g++ compiler 11
 - in a Docker container 4
 - game of chance 119
 - game of craps 120
 - game playing 113
 - game systems xxi

- garbage value 278
 - Gates, William xxxiii
 - GCC Docker container 13
 - gcd algorithm 596, **596**, 621
 - general class average problem 50
 - generalities 351
 - generalized numeric operations 619
 - generate algorithm 620
 - ranges version (C++20) **563**, 564
 - generate_n algorithm 620
 - ranges version (C++20) **563**, 565
 - generating values to be placed into elements of an array 162
 - generator coroutine **837**, 839
 - Fibonacci sequence 837
 - generator coroutine support library **836**
 - Sy Brand 837
 - `tl::generator` class template **837**
 - generator function 563
 - generic algorithms 558
 - generic lambda **176**, 177, 201, 395, 562
 - generic programming xxv, **137**
 - get function for obtaining a tuple member **681**
 - get member function of a `unique_ptr` 444
 - get member function of class template `future` 815
 - get pointer **244**
 - get_id function of the `std::this_thread` namespace (C++11) **772**
 - get_return_object function of a coroutine promise object **854**
 - getline function of the `string` header **95**
 - gets the value of 32
 - getting questions answered xxxvi
 - Git xliii
 - GitHub xxiv, xxxiii, xxxiv, xxxvi, xxxix, xliii
 - C++20 Standard Document xxxv
 - global 287
 - global function **103**
 - global module **725**
 - global module (C++20 modules) **725**
 - global module fragment **725**
 - global namespace scope 124, 125, 298, **719**, 720
 - global object constructors 298
 - global scope 298, 300
 - global variable 125, **125**, 125, 128, 133
 - GNU C++ xxxiii, xliii, 4
 - GNU C++ Standard Library Reference Manual xxxv
 - GNU Compiler Collection (GCC) Docker container xxv, 4, 13
 - GNU g++ xxv
 - GNU GCC 709
 - Godbolt, Matt
 - Compiler Explorer xxxix
 - `godbolt.org` xxxix, xl
 - Compiler Explorer website **498**
 - Goetz, Brian xl
 - golden mean (golden ratio) **143**
 - Google C++ Style Guide 493
 - Google Concurrency Library (GCL) concurrent containers 830
 - Google Logging Library (glog) 494
 - Google Search xl
 - goto elimination 40, 41
 - goto statement **40**
 - Grammarly xl
 - graph information 164
 - greater function object 604
 - greater_equal function object 604
 - greater-than operator 31
 - greater-than-or-equal-to operator 31
 - greatest common divisor 596
 - greedy evaluation 611, 836
 - greedy quantifier **263**
 - Grimm, Rainer xl
 - blog xxxv
 - grouping (operators) **31**
 - grouping not changed by overloading 424
 - groups.google.com/g/comp.lang.c++.xxxvi
 - `<gs1/gsl>` header 110
 - guard condition in the UML 44
 - guarding code with a lock **788**
 - Guidelines Support Library (GSL) **110**, 199
- ## H
- .h filename extension (header) 272
 - half-open range **178**, 518
 - handle on an object **291**
 - handle_contract_violation default contract violation handler **500**
 - has-a* relationship **308**, 337
 - hash (hashable keys) 533
 - hash bucket **552**
 - hash table 552
 - hash-table collisions 552, 553
 - hashable 537, 541
 - type requirements 533
 - hashing 533, **552**
 - Havender (deadlock prevention) 770
 - head of a queue 508
 - header 111, 286, 495
 - `<cstdlib>` 490
 - `<gs1/gsl>` 110
 - header (.h) **272**, 272

- header <numeric> 596
 - header of a function 107
 - header-only library 94, 110, 712
 - inline variable 679
 - header unit (C++20 modules) **712**, 713, 714, 721
 - compile a header 714
 - headers 236
 - <algorithm> 444, 452, 501, 523, 556, 620
 - <array> 155
 - <atomic> **816**
 - <barrier> (C++20) **823**
 - <cassert> **483**
 - <chrono> (C++11) 284, **761**
 - <cmath> 77, 103, 104, 105
 - <compare> 460
 - <concepts> **641**
 - <condition_variable> (C++11) **787**
 - <coroutine> (C++20) **854**
 - <cstdint> 60
 - <cstdint> (C++11) 207
 - <deque> **531**
 - <exception> **491**
 - <execution> (C++17) **762**
 - <filesystem> (C++17) **241**, 494
 - <forward_list> 527
 - <functional> **604**
 - <future> (C++11) **813**
 - <initializer_list> **443**
 - <iomanip> 53
 - <iostream> **23**
 - <latch> (C++20) **820**
 - <limits> 63
 - <list> **526**
 - <map> **539**, 541
 - <memory> **427**, 556, 621
 - <mutex> (C++11) **787**, 804
 - headers (cont.)
 - <numbers> **104**
 - <numeric> 556, **621**
 - <queue> **545**, 546
 - <random> (C++11) **113**
 - <ranges> **177**
 - <regex> **261**, 265
 - <semaphore> (C++20) **827**
 - <set> 533
 - <stack> **543**
 - <stdexcept> **472**, 491
 - <stop_token> (C++20) **805**
 - <string> **37**
 - <thread> (C++11) **771**
 - <tuple> **679**
 - <type_traits> 644, 701
 - <unordered_map> 539, 541
 - <unordered_set> 533, 537
 - <utility> **448**
 - <variant> 391
 - <vector> 180
 - heap 546, 599
 - max heap 599
 - min heap 599
 - heapsort
 - make_heap algorithm **600**
 - pop_heap algorithm **602**
 - push_heap algorithm **601**
 - sort_heap algorithm **601**
 - sorting algorithm **599**
 - helper function **292**
 - heterogeneous lookup (associative containers; C++14) **537**
 - hexadecimal integer 194
 - hide implementation details 274
 - hide names in outer scopes 124
 - hierarchical relationship 339
 - hierarchy
 - of employee types 341
 - of exception classes 490
 - high-level concurrency features 836
 - higher-order functions **175**
 - highest level of precedence 31
 - “highest” type 109
 - hold-and-wait condition 770
 - Hollman, Dr. Daisy xxxix
 - horizontal tab ('\t') 25
 - HTTPS protocol 148
 - hypot function **104**
 - hypotenuse
 - in three-dimensional space 104
 - of a right triangle 104
- ## I
- I/O completion 481
 - identifier **28**, 42, 125
 - IEEE 754 floating-point standard 500
 - if single-selection statement **31**, 42, 44
 - with initializer 85
 - if...else double-selection statement 42, **44**, 45, 52
 - with initializer 85
 - .ifc filename extension 718
 - ifstream 240, 243, 244
 - IGNORECASE regular expression flag **266**
 - image (Docker) **xlv**
 - immutable **174**
 - data **784**
 - data and thread safety 757
 - keys 509
 - string literal **217**
 - implement an interface 385
 - implementation inheritance 349, 391
 - implementation of a member function changes 297
 - implicit conversion **53**, 277, 463, 464
 - improper 463
 - user defined 463
 - via conversion constructors 464

- implicit first argument 315
- implicit handle 291
- import statement (C++20)
 - modules) **720**
 - existing header as a header unit **712**
- in-class initializer **285**, 325
- in-memory
 - formatting 247
 - I/O **247**
- in parallel **756**
- include guard **286**, 720
- #include <iostream> 23
- includes algorithm 620
 - ranges version (C++20) 589, **590**
- including a header multiple times 286
- inclusive_scan
 - algorithm 621
 - parallel algorithm (C++17) **766**
- increment 75
 - a control variable **70**, 71
 - a pointer 208
 - an iterator 517, 664
 - expression 87
 - operator 454
- indefinite postponement 768, **769**, 770
- indentation 45, 46
- index **155**
- indexed access 531
- indexed name used as an *rvalue* 436
- indirect base class **340**, 340
- indirection **192**
 - operator (*) **193**
 - triple 373
- indirectly reference a value **192**
- indirectly_copyable concept (C++20) 561
- indirectly_readable concept (C++20) 561
- indirectly_swappable concept (C++20) 583
- indirectly_writable concept (C++20) 561, 572
- inequality operator (!=) 431
- inf (negative infinity) 471
- inf (positive infinity) 471
- infer (determine) a variable's data type **172**
- infer a lambda parameter's type 176, 201, 562
- infinite loop 73, 74, 140
- infinite range **613**
- infinite sequence 836
- information hiding **19**, 274
- inherit **336**
- inherit members of an existing class **336**
- inheritance **19**, 308, 336, 337, 340
 - as an implementation detail 409
 - hierarchy **339**
 - implementation 391
 - interface **384**, 391
 - multiple **397**, 398, 399
 - public **341**
 - virtual **403**
- initial state in the UML **41**
- initial value of control variable **70**
- initial_suspend function of a coroutine promise object **854**
- initialization
 - once-time, thread-safe 815
 - std::call_once (C++11) **816**
 - std::once_flag (C++11) **816**
- initialize a pointer 192
- initializer **158**
- initializer list **158**
- initializer_list class
 - template **443**, 443, 511, 535
 - size member function 443
- <initializer_list> header **443**
- initializing
 - an array's elements to zeros and printing the array 156
 - multidimensional arrays 170
 - the elements of an array with a declaration 158
- inline 128
 - function **128**
 - keyword **128**
 - variable 320, 679
- inline_executor (concurrency) **845**
- inner block 124
- inner_product algorithm 621
- innermost pair of parentheses 31
- inplace_merge algorithm 620
 - ranges version (C++20) **588**
- input xxvi
- input and output stream iterators 514
- input from string in memory 112
- input iterator 515, 517, 560
- input/output (I/O) 103
 - header <iostream> **23**
 - library functions 112
- input sequence **514**
- input stream iterator 514
- input stream object (cin) **29**
- input_iterator concept (C++20) 559, 570, 587, 653
- input_or_output_iterator concept (C++20) 565
- input_range concept (C++20) 559, 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595

- inputting from strings in memory 247
- insert 507
 - at back of vector 518
- insert
 - function of associative container **536**
 - function of containers 513
 - function of `multimap` 540
 - function of `multiset` **534**
 - function of sequence container **524**
 - function of `set` 538
 - function of `string` **234**
- inserter function template **571**
- instance **18**
- instance variable 273, 281
- instantiate
 - class template 627
 - template **627**
- Instructor-Led Training with Paul Deitel xxxvii
- `int` & 129
- `int` data type 24, 28, 57, 109
 - operands promoted to `double` 53
- integer 24, **27**
 - arithmetic 416
 - `BigInteger` class **61**
 - division **30**, 50
 - promotion **53**
 - types, fixed-size 207
- integral concept **641**, 648
- integral constant expression **80**, 426
- integral expression 85
- inter-thread communication 814
 - `std::future` 814
 - `std::promise` **814**
- interest rate 75
- interface **284**, 363, 384
 - dependency 753
 - inheritance 364, **384**, 391
 - of a class **284**
 - to a hierarchy 364
- internal **175**
- internal iteration **161**
- internal linkage 719
- International Standards Organization (ISO) 2
- `invalid_argument` exception 235, **287**, 492
- invariant 495
 - class 495
- invoke function 610
- `<iomanip>` header 53, 111
- `ios::app` file open mode **241**
- `ios::ate` file open mode 241
- `ios::beg` seek direction **245**
- `ios::binary` file open mode 241
- `ios::cur` seek direction **245**
- `ios::end` seek direction **245**
- `ios::in` file open mode 241, **243**
- `ios::out` file open mode **241**
- `ios::trunc` file open mode 241
- `<iostream>` header **23**, 111
- `iota`
 - algorithm 596, **597**, 621
 - range factory (C++20) 613
 - range factory (C++20), infinite range **613**
- is-a* relationship (inheritance) **337**, 407
- `is_arithmetic` type trait 655
- `is_base_of` type trait 699
- `is_heap` algorithm 621
- `is_heap_until` algorithm 621
- `is_partitioned` algorithm 621
- `is_permutation` algorithm 620
- `is_sorted` algorithm **501**, 621
- `is_sorted_until` algorithm 621
- `isEmpty` member function of a stack 632
- `isnan` function of header `<cmath>` **256**
- ISO (International Standards Organization) 2
- Issue navigator 8
- `istream` class 244, 247
 - `seekg` function **244**
 - `tellg` function **245**
- `istream_iterator` **514**
- `istreamstring` class **247**, 248, 249
- `iter_swap` algorithm 582, **583**, 620
- iteration 43, 145, 147
 - of a loop 87
- iteration statement xxvi, **41**, **42**, 47
 - `do...while` 78, 79
 - `while` **47**, 70
- iteration terminates 47
- iterative solution **140**, 147
 - factorial 146
- iterator **506**
 - contiguous (C++17) 515
 - minimum requirements 558
 - nested type names 667, 669
 - pointing to the first element of the container 513
 - pointing to the first element past the end of container 513
 - read/write 668
 - read-only 665
 - `string` 568
 - type names 516
- iterator 510, 512, 513, 516, 536, 538
- iterator adaptor **570**
 - `back_inserter` **571**
 - `std::reverse_iterator` **672**

- iterator concepts (C++20)
 - 559
 - complete list 559
 - <iterator> header 112, 571
 - iterator operation 664
 - iterator_category nested
 - type in an iterator 667
 - .ixx filename extension **718**, 718
- J**
- Jacopini, G. 40
 - Java Platform Module System (JPMS) 746
 - join function of a
 - std::jthread **775**
 - joining a thread 844
 - Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards (2005) 493
 - JSON (JavaScript Object Notation) **326**, 327
 - array 326
 - Boolean values 326
 - cereal library 251
 - data-interchange format 326
 - false 326
 - JSON object 326
 - null 326
 - number 326
 - RapidJSON library 251
 - serialization **326**
 - string 326
 - true 326
 - JSONInputArchive (cereal library) 331
 - JSONOutputArchive (cereal library) 329
 - jthread class (C++20) **771**, 776
- K**
- Kalev, Danny Ph.D. xxxix
 - key 533
 - key-value pair **509**, 539, 540, 542, 552
- keyboard 29, 240
 - keys range adaptor (C++20) 612, **616**
 - keyword **24**, 42
 - auto **172**
 - break **83**
 - case **83**
 - class 137, **273**, 630
 - co_await (C++20) 834
 - co_return (C++20) 834
 - co_yield (C++20) 834
 - concept (C++20) **648**
 - const 115
 - constexpr **162**
 - continue **86**
 - default **83**
 - do 42, **78**
 - else 42
 - enum **123**
 - enum class **120**
 - explicit **277**, 464
 - for 42, **71**
 - if 42
 - inline **128**
 - namespace **719**
 - operator **423**
 - private **274**, 275
 - public **274**
 - static **123**, 124
 - switch 42
 - template **630**, 630
 - thread_local (C++11) **758**
 - throw 476
 - typename **137**, 630
 - unsigned **109**
 - while 42, **78**
 - Kühl, Dietmar xxxix, xl
- L**
- label in a switch **83**
 - lambda **176**, 557, 560
 - capture variables **257**, 456
 - expression **176**, 455
 - generic **176**, 201, 395
 - infer a parameter's type 176, 201, 562
 - lambda (cont.)
 - introducer **176**, 257, 456, 562
 - introducer [&] (capture by reference) **562**
 - introducer [=] (capture by value) **562**
 - templated (C++20) 636
 - templated 634
 - last-in, first-out (LIFO)
 - data structure 509, 543
 - order 629, 632
 - last member function of
 - span class template (C++20) **215**
 - latch (C++20) 820, **820**, 820, 821
 - <latch> header 113
 - late binding **358**
 - launch a long-running task
 - asynchronously 834
 - launch enum **814**
 - async 814
 - launch policy (multithreading) 814
 - lazily computed sequence
 - (generator) 835
 - lazy evaluation **178**, 611, 836
 - lazy pipeline 179
 - lcm algorithm 596, **597**, 621
 - leaf node in a class hierarchy **364**
 - least common multiple 597
 - left align (<) in string formatting **99**
 - left brace { } **24**
 - left fold
 - binary 689
 - unary 688
 - left justified 45
 - left-shift operator (<<) 416
 - left side of an assignment 93, 155, 302, 436
 - left stream manipulator **77**
 - left-to-right evaluation 31
 - left value 93
 - legacy code 425

- length member function of class `string` **37**
 - length of a string 217
 - `length_error` exception 228, **492**
 - `less` function object 604
 - less-than operator 31
 - less-than-or-equal-to operator 31
 - `less_equal` function object 604
 - `less<int>` **533**, 539
 - Levi, Inbal xxxix
 - lexicographical **226**
 - comparison 37, 419
 - sort 169
 - `lexicographical_compare` algorithm 620
 - ranges version (C++20) 566, **568**
 - `lexicographical_compare_three_way` algorithm 621
 - libraries
 - header-only 94
 - `miniz-cpp` 94
 - lifetime of an object 273
 - LIFO (last-in, first-out) 509, 543
 - order 632
 - “light bulb moment” 418
 - `<limits>` header 63, 112
 - linear running time **550**
 - linear search algorithm 551
 - linked list **507**
 - Linux
 - shell prompt 4
 - `list` class 518, **526**
 - `list` class template 508
 - `<list>` header 111, **526**
 - `list` member functions
 - `assign` **530**
 - `merge` 529
 - `pop_back` 530
 - `pop_front` **530**
 - `push_front` **527**
 - `remove` **531**
 - `list` member functions (cont.)
 - `sort` **528**
 - `splice` **528**
 - `swap` **530**
 - `unique` **530**
 - literal
 - character 261
 - digits 261
 - floating point **77**
 - live-code approach xxii
 - Live Instructor-Led Training
 - with Paul Deitel xxxvii
 - LL for `long long` integer literals **63**
 - load factor **552**
 - local automatic object 302
 - local variable 49, 126, 315
 - static 125
 - `<locale>` header 112
 - lock
 - an object 789, 790, 791
 - release **787**
 - `lock_guard` class (C++11) **791**
 - `log` function 104
 - `log10` function 104
 - logarithm 104
 - logarithmic running time **551**
 - logging 494
 - logging libraries
 - `Boost.Log` 494
 - `Easilylogging++` 494
 - `Google Logging Library (glog)` 494
 - `Loguru` 494
 - `Plog` 494
 - `spdlog` 494
 - logic error **32**
 - slicing **475**
 - `Logic_error` exception **491**
 - logical AND (`&&`) operator **88**, 90
 - in a constraint 642
 - logical complement operator, **!** **90**
 - logical function object **604**
 - logical negation, **!** **90**
 - logical operators xxvii, **88**, 90
 - logical OR (`||`) operator 88, **89**
 - in a constraint 642
 - `Logical_and` function object 604
 - `Logical_not` function object 604
 - `Logical_or` function object 604
 - Loguru logging library 494
 - `long` data type **60**, 60, 110
 - `long double` data type **50**, 77, 110
 - `long long` data type **60**, 60, 61, 110
 - LL for literals **63**
 - long-running task 834
 - loop
 - body 79
 - continuation condition **42**, 70, 71, 72, 78, 79, 87
 - continuation condition fails 146
 - counter 70
 - statement **42**
 - lossless data-compression algorithm **94**
 - lossy data-compression algorithm 94
 - `lower_bound` algorithm 620
 - ranges version (C++20) 592, **593**
 - `lower_bound` function of associative container **536**
 - lowercase letter 28, 112
 - “lowest type” 109
 - lvalue* (“left value”) **93**, 131, 155, 193, 302, 436, 454, 532
 - lvalues* as *rvalues* 93
- ## M
- m*-by-*n* array **170**
 - machine dependent 208

- machine learning 222, 250
- macro 111, **711**
 - preprocessor **104**
- magic numbers **162**
- main **24**
 - thread **775**
- “make your point” 119
- make_heap algorithm 620
 - ranges version (C++20) **600**
- make_pair function **540**
- make_tuple function **681**
- make_unique function **428**, 430, 446
- mangled function name 135
- manipulator 77
- map associative container 533
- map container class template 509
- <map> header 111, **539**, 541
- mapped values 533
- mapping in functional-style programming **179**, 611
- match_results class **265**
 - suffix member function of class match_results **267**
- math library 103, 111
- math library functions
 - ceil 104
 - cos 104
 - exp 104
 - fabs 104
 - floor 104
 - fmod 104
 - log 104
 - log10 104
 - pow 104
 - sin 104
 - sqrt 104
 - tan 104
- mathematical algorithms of the standard library 574
- mathematical constants 104
- mathematical special functions 105
- max algorithm **281**, 594, 620
- max heap 599
- max_element algorithm 620
 - ranges version (C++20) 574, **576**
- max_size member function
 - container 513
 - string **228**
- maximum function 105
- maximum integer value on a system 761
- maximum size of a string **228**, 228
- mdarray container (C++23) 173
- measures of central tendency **257**
- member function **18**
 - automatically inlined 288
 - call **19**
 - defined in a class definition 288
 - no arguments 288
 - parameter **107**
- member-initializer list **276**, 311, 399
- member object initializer 311
- member selection operator (.) 291, 292, 358, 429
- memberwise assignment **424**
- memory 16
 - address 192
 - allocate **425**, 425
 - consumption 373
 - deallocate **425**
 - footprint of exceptions 470
 - leak **417**, 426, 427, 431, 508
 - leak, preventing 429
 - management 103
 - utilization 553
- memory-access violation 508
- <memory> header 112, **427**, 556, 621
- memory-space/execution-time trade-off 552
- merge
 - algorithm 620
 - algorithm, ranges version (C++20) 584, **586**
 - member function of associative containers (C++17) 533
 - member function of list 529
- merge symbol in the UML **47**
- metacharacter (regular expressions) **261**
- metafunction 696
 - return value **697**
 - template argument **697**
 - type 697
 - value 697
- metaprogramming xxv
 - Meyer, Bertrand
 - design by contract **496**
- Microsoft xxxiv, 830
 - Visual C++ xxv
- Microsoft C++ language documentation xxxv
- Microsoft C++ Team Blog xxxv
- Microsoft modularized standard library 741
- Microsoft open-source C++ standard library 663
- Microsoft Parallel Patterns Library concurrent containers 830
- Microsoft Visual Studio
 - Community edition xliii, 4
- Microsoft Windows 82
- midpoint algorithm 621
- milliseconds object **761**
- min algorithm 594, **594**, 620
- min heap 599
- min_element algorithm 620
 - ranges version (C++20) 574, **576**
- minimum iterator requirements 558
- minimum requirements standard library algorithms 556

- miniz-cpp library 94
 - zip_file class 94, 96
 - zip_info class 97
- minmax algorithm 594, **595**, 620
 - ranges version (C++20) **595**
- minmax_element algorithm 620
 - ranges version (C++20) 574, **576**
- minus function object 604
- mismatch algorithm 620
 - ranges version (C++20) 566, **567**
- mismatch_result **567**
- mismatch_result for the mismatch algorithm **567**
- missing data 254
- missing values **253**
- mixed-type expression **109**
- Modern C++ xxi, 2, 190
 - do more at compile-time 163, 626
- modifiable data **757**
- modifiable *lvalue* 422, 436, 454
- modify a constant pointer 204
- modify address stored in pointer variable 204
- modular architecture of this book xxv
- modular standard library 740
 - Microsoft 740
- modular standard library (C++23) 746
- modules (C++20) **708**, 725
 - building a module with partitions 735
 - clang++ precompiled module (.pcm) file **722**
 - export a block of declarations 716
 - export a declaration **716**, 716, 719, 752
 - export a definition 752
- modules (C++20) (cont.)
 - export a namespace 717
 - export a namespace member 717
 - export followed by braces 752
 - export module **718**
 - export module declaration 752
 - filename extension .cpp 719
 - filename extension .cppm 719
 - filename extension .ifc 718
 - filename extension .ixx **718**, 718
 - filename extension .pcm 719
 - fmodules-ts compiler flag (g++) **714**
 - global module 752
 - global module fragment 752
 - header unit **712**, 752
 - IFC (.ifc) format 752
 - import a header file 752
 - import a module 752
 - import declaration **720**, 752
 - import existing header as a header unit **712**
 - improve compilation performance 713
 - interface 716
 - linkage 753
 - Microsoft modularized standard library 741
 - modular standard library (C++23) 746
 - modularized standard libraries 740
 - modularized standard library (Microsoft) 740
 - module declaration **718**, 718, 726, 753
- modules (C++20) (cont.)
 - module implementation partition unit **735**
 - module implementation unit **726**, 753
 - module interface 716
 - module interface partition unit 718, **732**, 733, 734, 735
 - module interface partition unit export module declaration 733
 - module interface unit **718**, 753
 - module interface unit (C++20 modules) **718**
 - module interface unit compile in clang++ 722
 - module interface unit compile in g++ 721
 - module name 718, 753
 - module partition 753
 - module purview **718**, 753
 - module unit **717**, 753
 - named **726**
 - named module 732, 753
 - named module purview 753
 - partition **732**, 753
 - partition rules 733
 - precompiled module interface 753
 - primary module interface unit **718**, 733, 734, 753
 - :private module fragment **731**
 - private module fragment 753
 - purview **718**
 - reachability **744**
 - reachable declaration 753
 - templates 719
 - visibility **744**
 - visible declaration 753
 - x c++-system-header compiler flag (g++) **714**

- modulus function object 604
 - modulus operator, % **30**
 - monetary formats 112
 - money
 - Boost.Multiprecision monetary library 75
 - Moore's law xxv, 2, **16**, 16, 617, 759
 - most derived class **405**
 - move **438**, 447
 - move algorithm 620
 - ranges version (C++20) **586**
 - move assignment operator xxviii, 279, 417, **431**, 447, 449, 513
 - move constructor xxviii, 278, 417, **431**, 438, 447, 448, 511, 513
 - move semantics xxviii, 417, 435, 513
 - move assignment operator 439
 - move constructor 438
 - std::move function **438**, 439
 - move_backward algorithm 620
 - ranges version (C++20) **586**
 - MoveAssignable 513
 - multi **759**
 - multi-core 557
 - architecture **618**, 759
 - processor **17**, 154, 174
 - systems xxix
 - multicore **618**
 - multidimensional array xxvi
 - multidimensional array **170**
 - multiline comment **23**
 - multimap associative container 509, 533, 539
 - multipass algorithms **515**
 - multiple inheritance **340**, 340, 397, 398, 399, 400, 401
 - demonstration 397
 - diamond inheritance **402**
 - multiple-selection statement **42**
 - multiple-source-file program
 - compilation and linking process 290
 - multiplication 30
 - compound assignment operator, *= 57
 - multiplies function object 604
 - multiset associative container 533
 - multiset container class template 509
 - multithreading 17, **757**
 - condition variable 789
 - launch enum **814**
 - std::async function template **814**
 - std::future class template 814
 - std::packaged_task function template **815**
 - std::shared_future class template 815
 - mutable (modifiable) data **757**, 784
 - mutable data **757**
 - mutating sequence algorithms 619, **619**
 - mutex class (C++11) **787**, 788
 - <mutex> header (C++11) 112, **787**, 804
 - mutual exclusion **784**, 787, 788, 827
 - necessary condition for deadlock 770
 - thread safety 757
 - MyArray class 430, 432, 663, 673
 - definition 441
 - definition with overloaded operators 441
 - test program 432
- ## N
- Nadella, Satya xxxiv
 - name handle 291
 - on an object 291
 - name mangling **135**
 - to enable type-safe linkage 135
 - name of an array **155**
 - named module (C++20 modules) **726**, 732
 - named requirements **513**
 - named return value optimization (NRVO) **437**, 456
 - namespace
 - keyword **719**
 - member 720
 - qualifier 720
 - scope **124**
 - std **24**
 - std::chrono 761
 - naming conflict 315, 719
 - NaN (not a number) 254
 - narrow_cast operator **110**
 - narrowing conversion **56**, 109, 110
 - braced initializer 109
 - explicit 110
 - natural language processing 222
 - natural logarithm 104
 - Navigator area (Xcode) 8
 - Navigators
 - Issue 8**
 - Project 8**
 - NDEBUG to disable assertions 483
 - near container **509**
 - negate function object 604
 - negative infinity (-inf) 471
 - nested
 - blocks 124
 - control statements **54**, 55
 - for statement 164, 173
 - if...else statement **45**
 - parentheses **31**
 - try blocks 479

- nested requirement in C++20
 - concepts 654, 656
 - nested type 510
 - names in containers 672
 - names in iterators 667
 - nested_exception 477
 - network message arrival 481
 - new
 - failure 487
 - failure handler 489
 - operator 425
 - <new> header 487
 - new operator
 - calls the constructor 425
 - placement version 425
 - returning nullptr on failure 489
 - throwing bad_alloc on failure 488
 - newline ('\n') escape sequence 25, 25, 216
 - next_permutation algorithm 621
 - no guarantee (of what happens when an exception occurs) 476
 - no preemption (necessary condition for deadlock) 770
 - no throw exception safety guarantee 477
 - node_type in an associative container 512
 - [[nodiscard]] attribute 292
 - noexcept keyword (C++11) 448, 477, 486, 497
 - non-const member function 307
 - on a const object 307
 - on a non-const object 307
 - non-type template parameter 672
 - non-virtual interfaces 376
 - nonconstant pointer to constant data 203
 - noncontiguous memory layout of a deque 531
 - nondeterministic
 - random numbers 114
 - seed 118
 - none_of algorithm 620
 - ranges version (C++20) 578, 581
 - non-member function to overload an operator 459
 - nonmodifiable lvalue 422
 - nonmodifying sequence algorithms 619, 620
 - non-module translation unit 725
 - non-parameterized stream manipulator 53
 - non-static member function 315, 463
 - non-virtual interface idiom (NVI) 338, 376
 - nonzero treated as true 92
 - not a number 84
 - not equal 31
 - not_equal_to function object 604
 - note in the UML 41
 - nothrow object 489
 - nothrow_t type 489
 - notify_all function of a std::condition_variable_any 805
 - notify_one function of a std::condition_variable 789, 790, 791, 799
 - NRVO (named return value optimization) 437, 456
 - nth_element algorithm 621
 - NULL 192
 - null character ('\0') 216
 - null in JSON 326
 - null pointer (0) 192, 194
 - null-terminated string 217
 - nullptr 482
 - on new failure 489
 - nullptr constant 192
 - number of arguments 107
 - number systems xxxi
 - <numbers> header 104
 - numbers in JSON 326
 - numeric algorithms 605, 621
 - <numeric> header 174, 556, 596, 621
 - numeric literal
 - with many digits 63
 - numeric_limits class template 63
 - max function 761
 - numerical data type limits 112
 - NVI (non-virtual interface idiom) 376
- ## O
- O(1) 550
 - O(log n) 551
 - O(n) 550
 - O(n²) 550
 - object
 - leaves scope 298
 - lifetime 273
 - of a class 17, 19
 - of a derived class 352, 354
 - of a derived class is instantiated 349
 - object-oriented analysis and design (OOAD) 20
 - object-oriented language 20
 - object-oriented programming (OOP) xxv, 20, 336
 - object's vtable pointer 376
 - objects contain only data 290
 - objects natural
 - case studies xxiv
 - Objects-Natural Approach xxiii, 2
 - octa-core processor 17
 - O'Dwyer, Arthur xxxv
 - blog xxxv
 - offset
 - from the beginning of a file 245
 - into a vtable 375

- ofstream class 240, 241, 242
 - open function **242**
- once_flag (C++11) **816**
- One Definition Rule (ODR) **711**, 712
- one-pass algorithm 515
- one-time, thread-safe initialization of an object 815
- one-to-many
 - mapping 509
 - relationship **539**
- one-to-one mapping 509, 541
- online forums xxxvi
- OOAD (object-oriented analysis and design) 20
- OOP (object-oriented programming) **20**, 336
- open a file
 - for input 241
 - for output 241
 - that does not exist 242
- open function of ofstream **242**
- open source
 - code xxxiii
 - community xxxiv
 - Microsoft C++ standard library 663
- open-source class libraries 61
- Open Web Application Security Project (OWASP) 327
- opened 239
- operand **25**, 29
- operating system xxi
 - device driver polymorphism 363
- operator xxvi
 - (predecrement/post-decrement) **58**
 - (prefix decrement/postfix decrement) 58
 - ! (logical negation) 88
 - ! (logical NOT) **90**
 - != (inequality) 31, 32
- operator (cont.)
 - ?: (ternary conditional) 47
 - () (parentheses) 31
 - * (multiplication) 30
 - * (pointer dereference or indirection) **193**, 194
 - *= (multiplication assignment) 57
 - / (division) 30
 - /= (division assignment) 57
 - && (logical AND) **88**, 89
 - % (remainder) 30
 - %= (remainder assignment) 57
 - + (addition) 29, 30
 - ++ (prefix increment/postfix increment) 58
 - ++ (preincrement/postincrement) 58
 - += (addition assignment) **57**, 224
 - < (less-than operator) 31
 - << (stream insertion) **24**, 30
 - <= (less-than-or-equal-to) 31
 - = (assignment) 29, 30
 - = (subtraction assignment) 57
 - == (equality) 31
 - > (greater-than) 31
 - >= (greater-than-or-equal-to) 31
 - >> (stream extraction) **29**
 - || (logical OR) 88, **89**
 - address (&) 194
 - arrow member selection (->) **292**
 - associativity **31**
 - co_await **849**
 - compound assignment 57, 59
 - conditional operator, ?: **47**
 - decrement operator, -- 58
- operator (cont.)
 - delete **425**
 - dot (.) **37**
 - grouping **31**
 - logical AND, && **88**, 90
 - logical complement, ! **90**
 - logical negation, ! **90**
 - logical operators **88**, 90, 91
 - logical OR, || 88, **89**
 - member selection (.) 291, 292
 - modulus, % **30**
 - narrow_cast **110**
 - new **425**
 - overloading **30**, 137
 - postfix decrement **58**
 - postfix increment **58**
 - precedence 30
 - precedence and grouping chart 35
 - prefix decrement **58**
 - prefix increment **58**
 - remainder, % **30**, 858
 - scope resolution (::) 287
 - sizeof **205**, 206
 - sizeof... **674**
 - static_cast **52**
 - that cannot be overloaded 423
 - that you do not have to overload 424
 - unary minus (-) 53
 - unary plus (+) 53
 - unary scope resolution (::) **133**
- operator
 - functions **423**
 - keyword **423**
- operator bool stream member function 242, 244
- operator overloading 182, **416**
 - addition assignment operator (+) 440
 - addition operator (+) 423, 424

- operator overloading (cont.)
 - binary operators 424
 - cast operator [454](#)
 - choosing member vs. non-member functions 458
 - commutative operators [459](#)
 - conversion operator [454](#)
 - copy assignment (=) 435, 446
 - copy assignment operator (=) [420](#)
 - decrement operators 454
 - equality operator (==) 435, 451
 - function call operator () [466](#)
 - increment operators 454
 - inequality operator 434, 452
 - is not automatic 423
 - member vs. non-member functions 459
 - operator[] 453
 - operator+ 423
 - operator++ 455
 - operator<< 457, 458
 - operator= 446
 - operator== 451
 - operator>> 457
 - postfix increment operator 455
 - preincrement operator (++) 455
 - rules and restrictions 424
 - self-assignment [421](#)
 - stream extraction operator >> 457
 - stream insertion and stream extraction operators 432, 433, 440
 - subscript operator 436, 453
 - operator[]
 - const version 453
 - non-const version 453
 - operator+ 423
 - operator<< 457, 458
 - operator= 446, 511
 - operator== 451, 566
 - operator>> 457
 - optimizing compiler 77
 - optional class (C++17) 191
 - <optional> header 113
 - order in which constructors and destructors are called 298, 300, 349
 - order in which operators are applied to their operands 144
 - order of evaluation 145
 - ordered associative container 508, [509](#), 509, 533
 - O'Reilly Online Learning
 - xxxvi
 - free trial [xlii](#)
 - ostream class 244
 - seekp function [244](#)
 - tellp function [245](#)
 - ostream_iterator [514](#)
 - ostream class [247](#), 451
 - out-of-bounds array elements 157
 - out of scope 127
 - out_of_bounds exception 431, 524
 - out_of_range exception [185](#), 224, 236, 492
 - header <stdexcept> [454](#)
 - outer block 124
 - outer for statement 173
 - outliers 260
 - output xxvi
 - output iterator 515, 517, 560
 - output sequence [514](#)
 - output stream 523
 - output to string in memory 112
 - output_iterator concept (C++20) 559, 564
 - output_range concept (C++20) 559, 564
 - outputting to strings in memory 247
 - overflow_error exception [492](#)
 - overhead of runtime polymorphism 373
 - overload set in overload resolution [637](#), 658
 - overloaded function definitions 134
 - overloaded parentheses operator 533
 - overloaded stream insertion operator << 457
 - overloading 30, [134](#)
 - concept based [652](#)
 - constructor 297
 - function templates 651
 - functions 651
 - resolution 651
 - overloading << and >> 137
 - overloading operators 137
 - overload-resolution rules 453
 - override a function [357](#)
 - override keyword 358, [361](#)
 - OWASP (Open Web Application Security Project) 327
- ## P
- P operation on Dijkstra semaphore 827
 - pack a tuple [681](#)
 - pair [536](#)
 - pair class template [679](#)
 - pair of braces {} 34
 - par execution policy of a parallel algorithm (C++17) [762](#), 763
 - par_unseq execution policy of a parallel algorithm (C++17) [763](#)
 - parallel algorithms 618
 - std::execution::par execution policy [762](#), 763

- parallel algorithms (cont.)
 - `std::execution::par_unseq` execution policy **763**
 - `std::execution::parallel_policy` class **763**
 - `std::execution::parallel_sequenced_policy` class **763**
 - `std::execution::sequenced_execution` policy **763**
 - `std::execution::sequenced_policy` class **763**
 - `std::execution::unseq` execution policy **763**
 - `std::execution::unsequenced_policy` class **763**
- parallel execution **764**
- parallel operations **756, 756**
- `parallel_policy` class (C++17) **763**
- `parallel_sequenced_policy` class (C++17) **763**
- parameter **107**
 - list **107**
- parameter pack **674, 683**
 - expansion **684**
 - variadic template **685**
- parameterized stream manipulator **53, 77**
 - quoted **246**
- parameterized type **629**
- parentheses operator (()) **31**
- parentheses to force order of evaluation **35**
- partial template specialization **703, 704**
- `partial_sort` algorithm **621**
- `partial_sort_copy` algorithm **621**
- `partial_sum` algorithm **596, 598, 621**
- partition **753**
- partition algorithm **621**
- partition in a C++20 module **732**
 - name **732, 733**
- `partition_copy` algorithm **621**
- `partition_point` algorithm **621**
- partitions
 - building a module with **735**
- partitions (C++20 modules rules) **733**
- pass-by-pointer **195**
- pass-by-reference **129, 191, 195, 196, 198**
 - with a pointer parameter used to cube a variable's value **196**
 - with pointer parameters **195**
 - with reference parameters **130, 195**
- pass-by-value **129, 130, 195, 197, 203**
- passing arguments by value and by reference **130**
- path (C++17) **241**
- Paul Deitel
 - Full-Throttle training courses xxxvii
 - Live Instructor-Led Training xxxvii
- payroll system using runtime polymorphism **363**
- `.pcm` (precompiled module) file in clang++ **719, 722**
- Pearson eText xxxvii
- Pearson Revel xxxvii
- percent sign (%) (remainder operator) **30**
- perform operations sequentially **756**
- performance issues with exceptions **474, 477, 482**
- performance tips xxiii
- performing operations concurrently **756**
- permutable concept (C++20) **575**
- pipeline in C++20 ranges **178**
- placeholder in a format string **66**
- placeholder type **854**
- placement `delete` **425**
- placement `new` **425**
- platform dependency **769**
- Plog logging library **494**
- plus function object **604**
- pointer xxvii, **190, 191, 208**
 - arithmetic **208**
 - arithmetic, machine-dependent **208**
 - comparison **210**
 - declared `const` **204**
 - dereference (*) operator **193, 194**
 - expression **208**
 - handle **291**
 - operators `&` and `*` **194**
 - to a function **373, 373**
 - to an implementation **385**
 - to dynamically allocated storage **445**
 - to `void (void *)` **210**
- pointer **510**
- pointer-based array xxvii, **190, 191**
- pointer-based string **190, xxvii, 190, 191, 216**
- pointer nested type in an iterator **667**
- poll analysis program **167**
- PolY class template (Facebook Folly library) **409**
- polymorphic behavior **358**
- polymorphic processing **220**
- polymorphic video game **350**

- polymorphically invoking
 - functions in a derived class 402
- polymorphism 85, 334, 376
 - compile-time (static) **408**, 410, 513, **628**, 629
 - runtime 337
- pop
 - member function of a stack 632
 - member function of container adapters **543**
 - member function of `priority_queue` 546
 - member function of `queue` 545
 - member function of `stack` 543
- `pop_back` member function of `list` 530
- `pop_front` 527, 532, 545
- `pop_heap` algorithm 620
 - ranges version (C++20) **602**
- position number **155**
- positive infinity (`inf`) 471
- post contract keyword (GNU C++ early access implementation) **499**
- postcondition **495**
 - violations 495
- postdecrement **58**
- postfix decrement operator **58**
- postfix increment operator **58**
- postincrement **58**, 58, 59
- postincrement an iterator 517
- `pow` function **77**, 104
- `pow` member function of class `BigInteger` **64**
- power 104
- power of 2 larger than 100 47
- `#pragma once` 284
- `#pragma once` directive **286**
- `pre` contract keyword (GNU C++ early access implementation) **499**
- precedence 30, 60, 75, 144
 - of arithmetic operators xxvi
- precedence chart 35
- precedence not changed by overloading 424
- precision **53**, 99
- precision of a floating-point value **50**
- precompiled module (`.pcm`) file (clang++) **722**
- precondition **495**
 - violations 495
- predecrement **58**
- predefined function objects 604
- predicate function **292**, 528, 567, 571, 573, 579, 580, 581, 582, 586, 589, 592
- preemptive scheduling **769**
- prefix decrement operator **58**
- prefix increment operator **58**
- preincrement **58**, 58, 59
- preincrement operator (`++`)
 - overloaded 455
- “prepackaged” functions 103
- preprocessor `xxx`
 - directives **23**
 - macro **104**
 - state 714
- `prev_permutation` algorithm 621
- prevent memory leak 429
- primary module interface unit (C++20 modules) **718**, 733, 734
- prime factorization 808
- prime numbers
 - University of Tennessee Martin Prime Pages website 809
- principal in an interest calculation 75
- principle of least privilege **125**, 203, 283, 306, 516
- print spooling **776**
- printing
 - line of text with multiple statements 26
 - multiple lines of text with a single statement 26
- `priority_emplace` member function of `queue` 546
- `priority_queue` adapter class **546**, 599, 600, 601
 - `emplace` function 546
 - `empty` function 546
 - `pop` function 546
 - `push` function 546
 - `size` function 546
 - `top` function **546**
- `priority_queue` container class template 509
- privacy 148
- `private`
 - access specifier **274**, 275
 - base class 407
 - base-class data cannot be accessed from derived class 346
 - inheritance 341, **406**
 - members of a base class 341
 - `static` data member 321
- `:private` module fragment (C++20 modules) **731**
- `private` virtual function 377
- `private` virtual member functions 409
- probability 114
- procedural programming xxv
- producer 757, **776**, 777
- producer–consumer relationship **776**
- profiling xxix, 759, 764
- program in the general 337
- program in the specific 337
- program termination 302
- program to an interface, not an implementation 383

- programming paradigms
 - functional-style xxv
 - generic xxv
 - metaprogramming xxv
 - object-oriented xxv
 - procedural xxv
 - programming tips
 - C++ Core Guidelines xxiii, xxxi
 - C++20 modules xxiii
 - common programming errors xxiii
 - performance tips xxiii
 - security best practices xxiii
 - software engineering observations xxiii
 - project 5, 8
 - Project** navigator 8
 - projection in C++20
 - `std::ranges` algorithms 567, 608
 - promise object (coroutines) **854**
 - `final_suspend` member function **854**
 - `get_return_object` member function **854**
 - `initial_suspend` member function **854**
 - `return_value` member function **854**
 - `return_void` member function **854**
 - `unhandled_exception` member function **854**
 - `yield_value` member function **855**
 - promotion **53**
 - promotion rules **109**
 - prompt **29**
 - property injection **386**
 - protected** **405**
 - base class 407
 - base class member function 406
 - data, avoid 406
 - inheritance 341, **406**, 407
 - protected** (cont.)
 - member of a class 405
 - virtual function 377
 - pseudorandom numbers **117**
 - `-pthread` compiler flag 771
 - public**
 - member of a subclass 405
 - method 287
 - public** access specifier **275**
 - public** base class 407
 - public** inheritance **341**, 406
 - public** keyword **274**
 - public** services of a class **284**
 - public** static
 - class member 321
 - member function 321
 - pure abstract class **363**, 384
 - pure specifier (= 0) for a virtual function **363**
 - pure virtual function **363**
 - purview (C++20 modules) **718**
 - push member function
 - container adapters **543**
 - `priority_queue` 546
 - `queue` 545
 - `stack` 543, 632
 - `push_back` member function
 - `vector` **186**, 520
 - `push_front` member function
 - `deque` 531
 - `list` **527**
 - `push_heap` algorithm 620
 - ranges version (C++20) **601**
 - put pointer **244**
- Q**
- quad-core processor 17
 - quadratic running time 550
 - quantifier
 - ? **264**
 - {*n*,} **264**
 - {*n*,*m*} **264**
 - * **263**
 - + **263**
 - greedy **263**
 - in regular expressions **262**
 - quantum **768**
- questions, getting answered xxxvi
 - `queue` **508**
 - `queue` adapter class **545**
 - back function **545**
 - `emplace` function 545
 - empty function 545
 - front function **545**
 - pop function 545
 - push function 545
 - size function 545
 - `queue` container class template 509
 - `<queue>` header 111, **545**, 546
 - quotation marks 24
 - quoted stream manipulator **246**
- R**
- race condition **783**
 - radians 104
 - RAII (Resource Acquisition Is Initialization) 417, **427**, 431, 482, 487, 493, 776, 790
 - raise to a power 104
 - random-access iterator 515, 516, 531
 - operations 517, 664
 - random access to elements of a container 508
 - `<random>` header 111, **113**
 - random integers in range 1 to 6 114
 - random number 117
 - generation xxvi
 - random-number generation
 - distribution **114**
 - engine **114**
 - `random_access_iterator` concept (C++20) 559, 569, 570, 575, 579, 600, 653
 - `random_access_range` concept (C++20) 559, 575, 579, 600, 601, 602

- random_device random-number source **118**, 123
- randomizing **117**
 - die-rolling program 118
- range (C++20) **177**, 507, 514
- range adaptor (C++20) **611**
 - all 612
 - common 612
 - counted 612
 - drop 612, **615**
 - drop_while 612, **615**
 - elements 612, 617
 - filter 612
 - keys 612, **616**
 - reverse 612, **614**
 - split 612
 - take 612, **613**
 - take_while 612, **614**
 - transform 612, 615
 - values 612, **616**
- range-based for statement **159**, 224
 - with initializer **161**
- range checking 224, 431
- range concept (C++20) 559
- range factory 613
- range factory (C++20)
 - iota 613
 - iota for an infinite range **613**
- range of elements 525
- range-v3 project 622
- range variable 160, 172
- ranges concepts (C++20) 559
- <ranges> header (C++20) 113, **177**
- ranges library (C++20) **177**, 253
- rapidcsv header-only library 251
 - Document class 252
 - GetColumn member function of class Document 252, 254
 - GetRowCount member function of class Document 255
- rapidcsv library 223
- RapidJSON library 251
- raw data 246
- raw string literal **249**
- Raz, Saar xxxix
- rbegin
 - member function of containers 512
 - member function of vector **522**
- reachability (C++20 modules) **744**
- read 665
- read and print a sequential file 243
- read data sequentially from a file 243
- readers and writers problem **804**
 - std::shared_mutex class (C++11) **804**
- ready thread state 768
- real-time systems xxi
- record 240
- recursion xxvi, **139**, 146, 147
 - step **140**, 144
- recursive
 - call **140**, 144
 - factorial 142
 - function **139**
 - solution 147
- reddit.com/r/cpp/ xxxvi
- reduce
 - algorithm 596, **597**, 621
 - parallel algorithm (C++17) **766**
- reduction **163**, 174, 175, 180, 597, 685
- refactor 338
 - payroll example 384
- reference 510
 - argument 195
 - parameter 129, **129**
 - to a constant 131
 - to a local variable 131
 - to an int 129
 - to private data 302
- reference nested type in an iterator 667
- <regex> header **261**, 265
- regex library
 - cmatch **265**
 - match_results class **265**
 - regex_constants **266**
 - regex_match function **261**
 - regex_replace function **265**
 - regex_search algorithm **265**
 - regex_search function **265**
 - smatch **265**
- regular expression 103, 247, **259**, 266
 - ? quantifier **264**
 - [] character class **262**
 - {n,} quantifier **264**
 - {n,m} quantifier **264**
 - * quantifier **263**
 - \ metacharacter **261**
 - \d character class **262**
 - \D character class **262**
 - \d character class **262**
 - \S character class **262**
 - \s character class **262**
 - \W character class **262**
 - \w character class **262**
 - + quantifier **263**
 - caret (^) metacharacter **263**
 - case insensitive 261
 - case sensitive 261
 - character class **261**, 262
 - ECMAScript grammar 260
 - escape sequence **262**
 - flavors 260
 - grammars 260
 - metacharacter **261**
 - regex_constants::icase 266
 - search pattern 260
 - validating data 260

- relational
 - function object **604**
 - operator **31**, **32**
- release
 - a lock **787**, **790**
 - a semaphore **827**
- release member function of `std::binary_semaphore` **829**, **830**
- relinquish the processor (yield) **819**
- remainder after integer division **30**
- remainder compound assignment operator, `%=` **57**
- remainder operator (`%`) **30**, **30**, **31**, **858**
- remove algorithm **620**
 - ranges version (C++20) **568**, **569**
- remove member function of `list` **531**
- remove_copy algorithm **620**
 - ranges version (C++20) **568**, **570**
- remove_copy_if algorithm **620**
 - ranges version (C++20) **568**, **572**
- remove_if algorithm **620**
 - ranges version (C++20) **568**, **571**
- remove_prefix member function of `string_view` **238**
- remove_suffix member function of `string_view` **238**
- rend
 - member function of containers **512**
 - member function of `vector` **522**
- repetition
 - counter controlled **52**
 - sentinel controlled **50**, **51**, **52**
- repetition statement **42**
 - `do...while` **42**
 - `for` **42**
 - `while` **42**, **49**, **52**
- replace `==` operator with `=` **92**
- replace algorithm **572**, **620**
 - ranges version (C++20) **572**, **572**
- replace member function of class `string` **232**, **233**
- replace_copy algorithm **620**
 - ranges version (C++20) **572**, **573**
- replace_copy_if algorithm **620**
 - ranges version (C++20) **572**, **574**
- replace_if algorithm **620**
 - ranges version (C++20) **572**, **573**
- representational **78**
- representational error **78**
- representational error in floating point **78**
- reproducibility xxxiv
- request_stop member function of a `std::jthread` **807**, **808**
- requirements **20**
- requires clause (C++20) **640**
- requires expression (C++20) **654**
- reserve member function of class `string` **230**
- reserved word **42**
 - `false` **44**
 - `true` **44**
- reset **547**
- resize member function of class `string` **230**
- Resource Acquisition Is Initialization (RAII) **417**, **427**, **482**, **487**, **493**
- resource leak **429**, **475**, **493**
- resource sharing **769**
- result (conurrencpp) **841**
- rethrow an exception **477**
- return a value **24**
- Return* key **29**
- return statement **25**, **108**, **140**
- return_value function of a coroutine promise object **854**
- return_void function of a coroutine promise object **854**
- returning a reference from a function **131**
- returning a reference to a private data member **302**
- reusable software components **17**
- reuse **18**, **37**, **272**
- Revel (Pearson) xxxvii
- reverse algorithm **620**
 - ranges version (C++20) **584**, **587**
- reverse range adaptor (C++20) **612**, **614**
- reverse_copy algorithm **620**
 - ranges version (C++20) **588**, **589**
- reverse_iterator **510**, **512**, **516**, **522**
- rfind member function of class `string` **231**
- right align `>` (C++20 text formatting) **99**, **100**
- right brace (`}`) **24**, **49**, **52**
- right fold
 - binary **689**
 - unary **688**
- right operand **25**
- right shift operator (`>>`) **416**
- right stream manipulator **77**
- right value **93**
- rightmost (trailing) arguments **133**
- robust application **469**

- rolling dice 115, 119
 - rotate algorithm 620
 - rotate_copy algorithm 620
 - round a floating-point number for display purposes 54
 - round-robin scheduling **769**
 - rounding numbers **54**, 78, 104
 - rows **170**
 - RSA Public-Key Cryptography algorithm **808**
 - RTTI (runtime type information) 409
 - Rule of Five (for special member functions) 444
 - Rule of Five defaults **444**
 - Rule of Zero (for special member functions) 279, 444, 665
 - rules of operator precedence **30**
 - run-length encoding 94
 - running* state 768
 - runtime (conurrencpp) **841**, 843
 - runtime concept idiom **408**
 - private virtual member functions 409
 - runtime polymorphism **337**
 - using class hierarchies 376
 - with virtual functions 358
 - runtime type information (RTTI) **409**
 - runtime_error class **472**, 480, 491, 492
 - what function 476
 - rvalue ("right value") **93**
 - rvalue ("right value") 131
 - rvalue reference (&&) 435, **438**, 439, 447, 448, 449
- S**
- SalariedEmployee class
 - header 367
 - implementation file 368
 - same_as concept (C++20) **649**
 - sample algorithm 620
 - savings account 75
 - schedule a task to execute 844
 - scheduling threads 769
 - scientific notation **53**
 - scope **124**, 719
 - class **124**
 - example 125
 - file **125**
 - function **124**
 - function parameter **124**
 - namespace **124**
 - scope of a variable **73**
 - scope resolution operator (::) 121, 124, 287, 321, 399, 634, 720
 - scoped enumeration (enum class) **120**
 - scoped_lock class (C++11) **791**
 - scraping 260
 - screen 23
 - screen-manager program 350
 - scrutinize data 287
 - search algorithm 620
 - search pattern (regular expressions) 260
 - search_n algorithm 620
 - searching 508, 578
 - arrays xxvi, **168**
 - second data member of pair **536**
 - secondary storage 16
 - secondary storage device **222**
 - security 148
 - best practices xxiii
 - flaws 157
 - seed
 - nondeterministic 118
 - the random-number generator **117**, 118
 - seek
 - direction **245**
 - get 244
 - put 244
 - seekg function of istream **244**
 - seekp function of ostream **244**
 - selection 43
 - selection statement xxvi, **41**, 42
 - if 42, 44
 - if...else 42, **44**, 45, 52
 - switch 42, 84
 - with initializer 85
 - self-assignment **449**
 - in operator overloading **421**
 - self-driving car 469
 - semaphore 826
 - acquire 827
 - release 827
 - <semaphore> header (C++20) 113, **827**
 - semicolon (;) **24**, 34
 - send a message to an object 19
 - sentinel (C++20 ranges) 525
 - sentinel-controlled iteration xxvi, 51, 52
 - sentinel value **50**, 52
 - separate interface from implementation **284**
 - seq execution policy (C++17) **763**
 - sequence **41**, 43, 170, 514
 - sequence container **508**, 508, 516, 518, 524, 528, 545
 - back function **524**
 - empty function **525**
 - front function **524**
 - insert function **524**
 - sequence of random numbers 117
 - sequenced_policy class (C++17) **763**
 - sequential file 240, 243, 246, 247
 - serialization xxvii
 - avoid language native serialization 327
 - pure data formats 327
 - security 327

- serializing data **326**
- set associative container 533, 537
- set container class template 509
- set function
 - validate data 279
- <set> header 111, 533, 537
- set_new_handler function **487**, 489, 490
- set operations of the standard library 590
- set_difference algorithm 620
 - ranges version (C++20) 589, **591**
- set_intersection algorithm 620
 - ranges version (C++20) 589, **591**
- set_symmetric_difference algorithm 620
 - ranges version (C++20) 589, **591**
- set_union algorithm 620
 - ranges version (C++20) **592**
- setprecision stream manipulator **53**
- setw parameterized stream manipulator **77**
- SFINAE (substitution failure is not an error) **410**, 658
 - obviated by C++20 concepts 410
- shadow 315
- shallow copy **445**
- Shape class hierarchy 340
- share data 757
- shared buffer 777
- shared mutable data 757, 784
- shared_lock class (C++11) **804**
- shared_mutex class (C++11) **804**
- <shared_mutex> header 112
- shared_ptr class 428
- shell prompt on Linux 4
- shift_left algorithm 620
- shift_right algorithm 620
- shifted, scaled integers 115
- short-circuit evaluation **90**, 642
- shrink_to_fit member function of classes vector and deque **522**
- shuffle algorithm 620
 - ranges version (C++20) 574, **575**
- side effect 129
 - of an expression **125**, 129, 145
- signal
 - a latch 820
 - operation on semaphore 827
- signal value **50**
- signature **108**, 135
 - of overloaded prefix and postfix increment operators 455
 - overriding a base-class virtual function 357
- SIMD (single instruction, multiple data) instructions 759
- simple condition 88
- simple requirement in C++20 concepts 654, **654**
- simulation
 - techniques xxvi
- sin function 104
- sine 104
- single-argument constructor 464, 465
- single-entry/single-exit control statements **43**
- single inheritance **340**, 399, 401, 402
- single instruction, multiple data (SIMD) instruction 759
- single-line comment **23**
- single-precision floating-point number **77**
- single quote 25
- single quote (') 216
- single-selection statement **42**
 - if 44
- single-threaded application 757
- single-use gateway 820
- singly linked list 508, 527
- six-sided die 114
- size
 - of a string 227
 - of a vector **519**
 - of an array 205
- size_global function 181
- size member function
 - array **155**
 - containers 513
 - initializer_list 443
 - priority_queue 546
 - queue 545
 - stack 543, 632
 - string_view **239**
 - vector **181**
- size_t type **157**, 205
- size_type 510
- sizeof operator **205**, 206
 - applied to an array name returns the number of bytes in the array 206
 - used to determine standard data type sizes 206
- sizeof... operator **674**
- sleep interval **768**
- sleep_for function of the std::this_thread namespace (C++11) **773**
- sleep_until function of the std::this_thread namespace (C++11) **773**
- sleeping thread **768**
- slicing (logic error) **475**
- small circles in the UML **41**
- smart pointer **427**
 - make_unique function template **428**, 430
 - unique_ptr 431

- smatch **265**
 - str member function **267**
- software engineering
 - information hiding **274**
 - observations xxiii
 - reuse 37, 272, 283
 - separate interface from implementation **284**
- software reuse 103, 630
- solid circle in the UML **41**
- solid circle surrounded by a hollow circle in the UML **41**
- solution 5
- Solution Explorer** 5
- Solution Explorer** in Visual Studio Community Edition 5
- sort algorithm **168, 169**, 459, 620, 759
 - ranges version (C++20) 578, **579**, 609, 610
- sort member function of list **528**
- sort_heap algorithm 620
 - ranges version (C++20) **601**
- sorting 508, 578
 - arrays xxvi
 - arrays **168**
 - order 580, 586
 - related algorithms 619
 - strings 112
- space–time trade-off **521**
- spaceship operator (<=>) xxviii, 459
- span class template (C++20) 191, **210**
 - back member function **214**
 - first member function **215**
 - front member function **214**
 - last member function **215**
 - subspan member function **215**
- header (C++20) 113, 191, **210**
- spdlog logging library 494
- special characters 28
- special member functions xxviii, **278**, 417, 431
 - constructor 275, 278
 - containers 511
 - copy assignment operator xxviii, 278, 417, **420**, 446
 - copy constructor xxviii, 278, 417, 446
 - destructor xxviii, 279, 298, 417
 - move assignment operator xxviii, 279, 417
 - move constructor xxviii, 278, 417
 - remove with = delete 444
 - Rule of Five 444
 - Rule of Zero 444
- specialized memory algorithms 621
- specialized memory operations 619
- spiral 143
- splice member function of list **528**
- splice_after member function of class template forward_list 528
- split range adaptor (C++20) 612
- pooling 777
- spurious wakeup **789**
- sqrt function of <cmath> header 104
- square function 110
- square root 104
- <sstream> header 112, 247, **247**
- stable_partition algorithm 621
- stable_sort algorithm 621
- stack 507
 - stack adapter class 509, **543**
 - emplace function 543
 - empty function 543
 - pop function 543
 - push function 543
 - size function 543
 - top function **543**
 - Stack class template 629, 632, 633
 - stack data structure 629, 630
 - <stack> header 111, **543**
 - stack overflow 140
 - stack unwinding **476**, 479, 487
 - stackful coroutine 840
 - stackless coroutine **840**
 - StackOverflow xxxiii, xxxvi
 - stackoverflow.com xxxvi
 - stale value **784**
 - Standard C++ Foundation xxxv
 - standard C++20 concepts by header **642**
 - standard concepts (C++20) 640
 - standard data type sizes 206
 - standard document (C++) xxxv
 - standard exception classes 492
 - bad_alloc **487**, 491
 - bad_cast **491**
 - bad_typeid **491**
 - exception **491**, 491
 - invalid_argument **492**
 - length_error **492**
 - logic_error **491**
 - out_of_range **492**
 - overflow_error **492**
 - runtime_error **472**, 480, 491, 492
 - underflow_error **492**
 - standard input stream object (cin) 240
 - standard library 103
 - class string 35, 418
 - deque class template 532

- standard library (cont.)
 - exception hierarchy 490
 - headers 112
 - list class template 527
 - map class template 541
 - multimap class template 539
 - multiset class template 534
 - priority_queue adapter class 547
 - queue adapter class templates 545
 - set class template 538
 - stack adapter class 543
 - vector class template 519
- standard library algorithms
 - minimum requirements 556
- standard library exception
 - filesystem_error 494
- standard library exception hierarchy 490
- standard output object (cout) 24
- standard output stream object (cout) 240
- Standard Template Library (STL) 506
- Start Window 4
- Start Window in Visual Studio Community Edition 4
- starts_with member function of class string (C++20) 38
- starts_with member function of string_view 239
- starvation 769
- state dependent 777
- statement 24
 - break 83, 86
 - continue 86
 - control statement 41, 43
 - control-statement nesting 43
 - control-statement stacking 43
- statement (cont.)
 - do...while 42, 78
 - double selection 42
 - empty 46
 - for 42, 71, 75, 77
 - if 31, 42, 44
 - if...else 42, 44, 45, 52
 - iteration 41, 47
 - looping 42
 - multiple selection 42
 - nested control statements 54
 - nested if...else 45
 - repetition 42
 - return 25
 - selection 41, 42
 - single selection 42
 - spread over several lines 34
 - switch 42, 80, 84
 - throw 287
 - try 185
 - while 42, 47, 49, 52, 70
- static binding 358
- static code analysis tools xxxii
 - clang-tidy xxxii, xlvi
 - cppcheck xxxii, xlvi
- static data member 320, 321
 - save storage 320
 - tracking the number of objects of a class 323
- static keyword 123, 124
- static local object 299, 301, 302
- static local variable 125, 127, 564
 - thread-safe initialization 815, 816
- static member 321
- static member function 321
- static polymorphism 408, 410, 629
- static polymorphism (compile-time) 628
- static_assert declaration 659
- static_cast operator 52, 60, 92
- statistics
 - measures of central tendency 257
- std namespace 24
- std::add_const metafunction 703
- std::advance function 652
- std::as_const function (C++17) 676
- std::async (C++11) 808
- std::async function template 814
- std::atomic class template 817
- std::atomic type 817
- std::atomic_ref class template (C++20) 817, 820
- std::barrier (C++20) 820, 823
- std::binary_semaphore (C++20) 827
 - acquire member function 829, 830
 - release member function 829, 830
- std::call_once (C++11) 816
- std::call_oncedefault para font> (C++11) 816
- std::chrono namespace 761
- std::chrono::duration class 761
- std::cin 29
- std::condition_variable class
 - notify_one function 789, 790, 791, 799
- std::condition_variable class (C++11) 787
- std::condition_variable_any class (C++11) 805
 - notify_all function 805

- `std::counting_semaphore` (C++20) **827**
- `std::cout` 24
- `std::distance` function **652**
- `std::execution::par` execution policy (C++17) **762**, 762, 763
- `std::execution::par_unseq` execution policy (C++17) **763**
- `std::execution::parallel_policy` class (C++17) **763**
- `std::execution::parallel_sequenced_policy` class (C++17) **763**
- `std::execution::seq` execution policy (C++17) **763**
- `std::execution::sequenced_policy` class (C++17) **763**
- `std::execution::unseq` execution policy (C++17) **763**
- `std::execution::unsequenced_policy` class (C++17) **763**
- `std::floating_point` concept **641**, 648
- `std::future` class template 814
- `std::hash` 533
- `std::initializer_list` class template **443**
- `std::integral` concept **641**, 648
- `std::invoke` function 610
- `std::jthread` (C++20)
 - `join` function **775**
 - `request_stop` function **807**, 808
- `std::latch` (C++20) **820**
 - `count_down` member function **821**
 - `wait` member function **821**, 821
- `std::launch` enum
 - `async` 814
 - `deferred` 814
- `std::lock_guard` class (C++11) **791**
- `std::move` function **438**, 439
- `std::mutex` class (C++11) **787**, 788
- `std::numeric_limits::max()` 761
- `std::once_flag` (C++11) **816**
- `std::optional` class (C++17) 191
- `std::packaged_task` function template **815**
- `std::promise` (C++11) **814**
- `std::ranges` namespace 507, 556
 - `all_of` algorithm 578, **580**
 - `any_of` algorithm 578
- `std::ranges` namespace (C++20) **525**, 560, 561, 563, 566, 568, 572, 574, 578, 582, 584, 588, 589, 592, 594, 599
- `std::ranges::count_if` algorithm (C++20) **257**, 258
- `std::ranges::distance` algorithm 663
- `std::reverse_iterator` iterator adaptor **672**
- `std::same_as` concept (C++20) **649**
- `std::scoped_lock` class (C++17) **791**
- `std::shared_future` class template 815
- `std::shared_lock` class (C++11) **804**
- `std::shared_mutex` class (C++11) **804**
- `std::size` global function 181
- `std::stop_callback` (C++20) **808**
- `std::stop_source` for cooperative cancellation (C++20) **807**
- `std::stop_token` for cooperative cancellation (C++20) **807**
 - `stop_requested` function **807**
- `std::string_literals` **168**
- `std::this_thread` namespace
 - `yield` function 819
- `std::this_thread::get_id` function (C++11) **772**
- `std::this_thread::sleep_for` function (C++11) **773**
- `std::this_thread::sleep_until` function (C++11) **773**
- `std::thread` class (C++11) **771**
- `std::thread::id` **772**
- `std::unique_lock` class (C++11) **788**, 789
- `std::variant` class template **391**
 - for runtime polymorphism **391**
- `std::visit` standard library function **391**, 395, 396
- `std::jthread` (C++20) **771**, 771, 776
- `std.core` in the Microsoft modularized standard library 740
- `std.filesystem` in the Microsoft modularized standard library 740
- `std.memory` in the Microsoft modularized standard library 740

- std. regex in the Microsoft modularized standard library 740
- std. threading in the Microsoft modularized standard library 741
- <stdexcept> header 112, 472, 491
 - out_of_range 454
- steady_clock class 761
- sticky setting 54, 77
- STL (Standard Template Library) 506
- stod function 235
- stof function 235
- stoi function 235
- stol function 235
- stold function 235
- stoll function 235
- stop_requested member function of a
 - std::stop_token 807
- <stop_token> header 113
- <stop_token> header (C++20) 805, 820
- stoul function 235
- stoull function 235
- str member function of an smatch 267
- str member function of class ostream 247, 248
- stream extraction operator >> 29, 34, 137, 416, 456
- stream input/output 23
- stream insertion operator << ("put to") 416
- stream insertion operator << 25, 29, 137, 243, 456
- stream manipulator 77
 - boolalpha 37
 - quoted 246
- stream manipulators 53
 - fixed 53
 - left 77
 - right 77
 - setprecision 53
 - setw 77
- stream of characters 24
- streaming 757
- <string> header 112
- string 168, 616
 - C style 190
 - pointer based 190
 - processing xxvii
- string
 - iterators 568
- string built-in type in JSON 326
- string class 35, 36, 112, 273, 417, 418, 509
 - assignment 223
 - assignment and concatenation 223
 - at member function 422
 - concatenation 223
 - empty member function 38, 419
 - ends_with member function (C++20) 38
 - find functions 230
 - find member function 230
 - insert functions 234
 - insert member function 234
 - length member function 37
 - starts_with member function (C++20) 38
 - subscript operator [] 224
 - substr member function 421
- string concatenation 38, 685
- string concatenation assignment 420
- string formatting 65
 - C++20 65, 66, 67
- <string> header 37
- string literal 24
 - raw string literal 249
- string object literal 168, 420, 616, 635
- string of characters 24
- string processing 103
- string stream processing 247
- string_literals 168
- string_view class (C++17) 190, 236, 274
 - find member function 239
 - remove_prefix member function 238
 - remove_suffix member function 238
 - size member function 239
 - starts_with member function 239
- <string_view> header (C++17) 236
- string::npos 231
- strings as full-fledged objects 216
- strong encapsulation 722, 743
- strong exception guarantee 446
 - copy-and-swap idiom 477
- strong exception safety guarantee 477
- Stroustrup, Bjarne
 - website xxxv
- struct 324
- structured binding (C++17) 595
 - unpack elements 577
- structured binding declaration 577
- structured programming 40, 88
- student-poll-analysis program 167
- subclass 336
- submit function of a concurrentcpp executor 844
- subobject of a base class 402
- subproblem 140
- subscript 155
- subscript operator 532
 - map 541, 542

- subscripted name used as an *rvalue* 436
 - subscripting 531
 - subspan member function of `span` class template (C++20) **215**
 - substitution cipher **148**
 - `substr` 227
 - `substr` member function of class `string` **226**, 421
 - substring of a `string` 226
 - subtract one pointer from another 208
 - subtraction 30, 31
 - subtraction compound assignment operator, `--` 57
 - sufficient conditions for deadlock 770
 - suffix member function of class `match_results` **267**
 - sum of the elements of an array 163, 174
 - superclass **336**
 - survey 166, 168
 - suspend a coroutine's execution 839
 - `suspend_always` (coroutines) **854**
 - `suspend_never` (coroutines) **854**
 - suspension point (coroutines) **855**
 - Sutter, Herb 376
 - blog **482**, 496
 - ISO C++ Convener **496**
 - Sutter's Mill Blog xxxv
 - `swap`
 - member function of class `unique_ptr` 459
 - standard library function 459
 - `swap` algorithm **583**
 - `swap` member function
 - of containers 513
 - of `list` **530**
 - `swap` member function of class `string` **227**
 - `swap_ranges` algorithm 582, 620
 - ranges version (C++20) **583**, 584
 - swapping strings 227
 - switch logic 85
 - switch multiple-selection statement 42, **80**, 84
 - case label 83
 - controlling expression **83**
 - default case **83**, 84
 - switch with initializer 85
 - synchronization **784**, 785
 - synchronization point 820
 - synchronized block of code 787
 - synchronized threads **757**
 - synchronous error **481**
 - syntax coloring conventions in this book xxxiii
 - `system_clock` class **761**
- T**
- Tab* key 24
 - tab stop 25
 - table of values **170**
 - tabular format 157
 - tag dispatch **411**, 658
 - obviated by C++20 concepts **411**
 - tail of a queue 508
 - tails 114
 - take range adaptor (C++20) 612, **613**
 - `take_while` range adaptor (C++20) 612, **614**
 - `tan` function 104
 - tangent 104
 - task (conurrencpp) **841**
 - task for asynchronous operations 836
 - `tellg` function of `istream` **245**
 - `tellp` function of `ostream` **245**
 - template definition 138
 - template function 138
 - template header **630**
 - template instantiations **137**
 - template keyword **137**, 630
 - template metaprogramming xxv
 - template metaprogramming (TMP) xxv, 628, **693**
 - metafunction 696
 - Turing complete 694
 - type metafunction 697
 - value metafunction 697
 - template parameter **630**
 - template parameter list **137**
 - templated lambda (C++20) 636
 - templated lambda expression 634
 - templates
 - compile-time code generation 410
 - constraints 411
 - deduction guide 673
 - default type argument for a type parameter **678**
 - defining in C++20 modules 719
 - partial specialization **703**
 - requirements for a type 410
 - type argument 630
 - variable template **678**
 - temporary value 52, 110
 - terminate a program 489
 - terminate normally 242
 - terminate standard library function **480**
 - terminate successfully 25
 - terminated state **769**
 - terminating condition 141
 - terminating right brace (`}`) of a block 124
 - termination
 - `abort` function **480**, 489
 - `exit` function 489
 - `terminate` function **480**
 - termination condition 157

- termination housekeeping **298**
- termination model of exception handling **475**
- termination test 146
- ternary conditional operator (`?:`) 145
- ternary operator **47**
- test 548
- test characters 112
- text editor 243
- text formatting 98
 - C++20 **65**, 66, 67
 - format specifier 99
- text-printing program 23
- the cloud 326
- this pointer **315**, 316, 324, 447, 450
 - used explicitly 315
 - used implicitly and explicitly to access members of an object 315
- thread **757**
 - exception 771
 - of execution **757**
 - scheduling **768**, 782
 - state **767**
 - synchronization **784**
- thread class (C++11) **771**
- thread-coordination primitives 816
- thread-coordination types 820
- `<thread>` header 112, **771**
- thread launch policy 814
- thread-local storage
 - thread safe 758
- thread pool **841**
- thread safe **757**, 783, 787
 - one-time initialization 815
 - atomic type 757
 - immutable data 757
 - linked data structures 819
 - mutual exclusion 757
 - thread local storage 758
- thread scheduler **769**
- thread states
 - blocked **769**
 - born **768**
 - ready **768**
 - running **768**
 - terminated **769**
 - timed waiting **768**
 - waiting **768**
- thread synchronization
 - coordination types 820
- thread_executor (concurrency) **844**
- thread_local storage class (C++11) **758**
- thread_pool_executor (concurrency) **841**, 844
- thread::id **772**
- C++20 xxviii, 113, 279, 459, **460**, 511
- three-way comparison operator (`<=>`) xxviii, 113, 279, 459, **460**, 511
- throw 476, 491
 - standard exceptions 491
- throw an exception **184**, 185, 287, 287, 474
 - from a constructor 484
- throw point **474**, 480
- tightly coupled **383**
- tilde character (`~`) 298
- time and date utilities 761
- Time class
 - constructor with default arguments 293
 - definition 285
 - definition modified to enable cascaded member-function calls 317
 - member-function definitions 286
 - member-function definitions, including a constructor that takes arguments 294
- timed waiting* thread state 768
- timer for performing a task in the future 836
- times 103
- timeslice **768**, 769
- Titanic* disaster dataset 223, 253
- `tl::generator` class template (generator library) **837**
- TMP (template metaprogramming) 628, **693**
- `to_array` function of header `<array>` (C++20) 191, **201**
- `to_string` function **235**
- token 246
- top member function
 - of `priority_queue` **546**
 - of `stack` **543**
- top member function of a `stack` 631
- top of a `stack` 507
- topical xxi
- trailing requires clause **649**
- trailing return types **563**
- transaction processing 539
- transform algorithm 620
 - ranges version (C++20) 574
- transform range adaptor (C++20) 612, 615
- `transform_exclusive_scan` algorithm 621
- `transform_exclusive_scan` parallel algorithm (C++17) **766**
- `transform_inclusive_scan` algorithm 621
- `transform_inclusive_scan` parallel algorithm (C++17) **766**
- `transform_reduce` algorithm 621
- `transform_reduce` parallel algorithm (C++17) **766**
- transforming data 260
- transition arrow in the UML **41**

- transition from the preprocessor to modules 712
 - translation look-aside buffers (TLBs) 553
 - translation unit 679, **710**, 713, 732
 - non-module 725
 - part of a module 717
 - treat warnings as errors 131
 - trigonometric cosine 104
 - trigonometric sine 104
 - trigonometric tangent 104
 - triple indirection 373
 - trivially copyable type **819**, **820**
 - true 32, **44**
 - truncate **30**, 241
 - truncate fractional part of a calculation 50
 - truncate fractional part of a `double` 109
 - truth tables for logical operators **88**, 89
 - try block **185**, 477, 480
 - expiration 475
 - nested 479
 - try statement **185**
 - tuple
 - pack **681**
 - unpacking **681**
 - tuple class template **679**
 - getting a tuple member **681**
 - `make_tuple` function **681**
 - `<tuple>` header (C++11) 111, **679**
 - Turing complete 694
 - two-dimensional array **170**
 - type alias 680
 - type argument 430, 604, 630
 - type category 648
 - type erasure **409**
 - type-erasure-based runtime polymorphism 409
 - type metafunction 697
 - predefine 704
 - type name, alias 394
 - type of the `this` pointer 315
 - type parameter **137**, 138, 630
 - type requirement in C++20 concepts 640, 654, **655**
 - type-safe linkage **135**
 - type-safe union 394
 - type trait 628
 - `is_base_of` 699
 - value member 646
 - `<type_traits>` header 701, 644
 - `typeid` 491
 - `<typeinfo>` header 112
 - typename keyword **137**, 630
 - typename... in a variadic template **683**
- ## U
- Ubuntu Linux 7
 - in the Windows Subsystem for Linux 7
 - UML (Unified Modeling Language)
 - activity diagram **41**, 41, 47, 79
 - arrow 41
 - diamond 44
 - dotted line **42**
 - final state **41**
 - guard condition **44**
 - merge symbol **47**
 - note **41**
 - solid circle **41**
 - solid circle surrounded by a hollow circle 41
 - UML class diagram **340**
 - unary left fold **686**, 688
 - unary minus (-) operator 53
 - unary operator **53**, 90, 193
 - unary operator overload 424
 - unary plus (+) operator 53
 - unary predicate function 571, 573
 - unary right fold **686**, 688
 - unary scope resolution operator (::) **133**
 - uncaught exception 479, 480
 - unconstrained function template 637
 - undefined behavior **63**, 194, 426, 445
 - division by zero 471
 - undefined value 278
 - `underflow_error` exception **492**
 - underlying data structure 546
 - underscore (_) 28
 - `unhandled_exception`
 - function of a coroutine promise object **854**
 - Unicode character set 85
 - `uniform_int_distribution` **114**
 - unincremented copy of an object 455
 - union **394**
 - unique algorithm 620
 - ranges version (C++20) 584, **586**
 - unique keys 533, 537, 541
 - unique member function of `list` **530**
 - `unique_copy` algorithm 620
 - ranges version (C++20) 588, **589**
 - `unique_lock` class (C++11) **788**, 789
 - `unlock` function **790**
 - `unique_ptr` 431
 - `unique_ptr` class (C++11) 428, **428**, 431
 - built-in array **430**
 - create with `make_unique` function template 446
 - `get` member function 444
 - `swap` member function 459
 - universal-time format 287
 - University of Tennessee Martin Prime Pages website 809
 - UNIX 82, 242
 - `unlock` function of a `unique_lock` **790**

- unordered associative containers **508, 509, 511**, 533
- `unordered_map` associative container class template 509, 533, 541
- `<unordered_map>` header 111, 539, 541
- `unordered_multimap` associative container class template 509, 533, 539
- `unordered_multiset` associative container class template 509, 533
- `unordered_set` associative container class template 509, 533, 537
- `<unordered_set>` header 111, 533, 537
- unpack elements (C++17) structured binding **577**
- unpacking a tuple **681**
- Unruh, Erwin 694
- unseq execution policy (C++17) **763**
- `unsequenced_policy` class (C++17) **763**
- unsigned char data type 110
- unsigned data type 110
- unsigned int data type 110
- unsigned integer types **109**
- unsigned long data type 110
- unsigned long int data type 110
- unsigned long long data type 110
- unsigned long long int data type 110
- unsigned short data type 110
- unsigned short int data type 110
- unwinding the function call stack 479
- update records in place 246
- `upper_bound` algorithm 620
 - ranges version (C++20) 592, **593**
- `upper_bound` function of associative container **536**
- uppercase letter 28, 112
- user-defined function 105
- user-defined type 120, 121, **272**, 462
- using a dynamically allocated `ostringstream` object 247
- using a function template 137
- using arrays instead of `switch` 165
- using declaration **33**, 720
 - in headers 274
- using declaration to create an alias for a type **394**, 680
- using directive **33**, 720
 - in headers 274
- using enum statement 124
- using function swap to swap two strings 227
- using standard library functions to perform a heapsort 600
- using virtual base classes 403
- Utilities** area (Xcode) 8, 9
- utility function **292**
- `<utility>` header 112
 - exchange function **448**
- V**
- V operation on semaphore 827
- validate a first name 262
- validate data in a `set` function 279
- validating data (regular expressions) 260
- value initialization **159**, 199, 278, 325, 426, 682
 - memory 443
 - objects 426
 - rules 426
- value member of a type-trait class 646
- value metafunction 697
- value of an array element **155**
- `value_type` 510
 - nested type in an iterator 667
- values range adaptor (C++20) 612, **616**
- variable **27**
- variable scope 73
- variable template (C++14) 646, **678**
- variadic function template 686
 - compile-time recursion 682
 - typename... **683**
- variadic template 628, 674, **679**
 - parameter pack **674**, 683
 - `sizeof... operator` **674**
- variadic template parameter pack 685
- `<variant>` header 391
- variant standard library class template 391
- vector 631
 - capacity 519
- vector class 180, 482
- vector class template **154**, 508, 519
 - capacity function **519**
 - `cbegin` function **522**
 - `crend` function **522**
 - erase member function 569
 - `push_back` function 520
 - `push_back` member function **186**
 - `push_front` function 520
 - `rbegin` function **522**
 - `rend` function **522**
 - `shrink_to_fit` member function **522**
- vector class template element-manipulation functions 523

- vector hardware operations 759
 - <vector> header 111, 180
 - vector mathematics **618**, **759**
 - vectorized execution 764
 - version control tools xxxiv
 - video streaming 795
 - videos
 - C++20 Fundamentals LiveLessons xxxvii
 - view (C++20) **177**, 507, 611
 - all range adaptor 612
 - common range adaptor 612
 - composable **177**
 - composing 611
 - counted range adaptor 612
 - drop range adaptor 612, **615**
 - drop_while range adaptor 612, **615**
 - elements range adaptor 612, 617
 - filter range adaptor 612
 - iota range factory 613
 - iota range factory for an infinite range **613**
 - keys range adaptor 612, **616**
 - reverse range adaptor 612, **614**
 - split range adaptor 612
 - take range adaptor 612, **613**
 - take_while range adaptor 612, **614**
 - transform range adaptor 612, 615
 - values range adaptor 612, **616**
 - view into a container 210
 - viewable_range (C++20) **611**
 - views of contiguous container elements 210
 - views::filter **178**, 179
 - views::iota **178**
 - Vigenère secret key cipher **147**, 148, 149, 150
 - violation handler (contracts) **503**
 - default **500**
 - virtual base class 402, 403
 - virtual destructor **361**
 - virtual function 337, **357**, 357, 373, 375, 402
 - as an internal implementation detail 382, 412
 - call 375
 - call illustrated 374
 - overhead 373
 - private 377, 409
 - protected 377
 - table (*vtable*) **373**
 - "under the hood" 373
 - virtual inheritance **403**
 - virtual memory 488, 490
 - Virtuality (paper) 376
 - visibility (C++20 modules) **744**
 - visit standard library function **391**, 395, 396
 - visitor pattern **411**
 - Visual C++ xxii
 - Visual C++ compiler xlv
 - Visual Studio Community Edition xliii, 4
 - Command Prompt window 6
 - Create a New Project** dialog 5
 - Create a New Project-Configure your new project** 5
 - Empty Project** template 5
 - Solution Explorer** 5
 - Start Window** 4
 - void * **210**
 - void return type **107**, 108, 109
 - volume of a cube 128
 - vtable* **373**, 376
 - vtable pointer 376
 - vtable* pointer 376
- ## W
- wait-for (necessary condition for deadlock) 770
 - wait function of a condition_variable **789**
 - wait member function of a std::latch **821**
 - wait operation on semaphore 827
 - waiting thread 790
 - state 768
 - "walk off" either end of an array 430
 - Wall GNU g++ compiler flag 93
 - warnings
 - treat as errors 131
 - weak_ptr class (C++11) 428
 - weakly_incrementable concept (C++20) 561
 - web service 326
 - Welcome to Xcode** window 8
 - what virtual function of class exception **186**, 475, 476, 488
 - when_all function (concurrency library) **848**
 - when_any function (concurrency library) **849**
 - while iteration statement 42, **47**, 49, 52, 70
 - whitespace characters **23**, 24
 - whole number 27
 - Williams, Anthony xxxix
 - Windows 82
 - Windows Subsystem for Linux (WSL) xxvi, **xlv**, 7
 - word character **262**
 - worker_thread_executor (concurrency) **845**
 - workflow **41**
 - workspace window 8
- ## X
- x c++-system-header compiler flag (g++) **714**

- x86-64 gcc (contracts) 498
- Xcode xliii
 - Debug** area 9
 - Editor** area 8, 9
 - Navigator** area 8
 - Utilities** area 8, 9
 - Welcome to Xcode** window 8
- Xcode navigators
 - Issue 8**
 - Project 8**
- Xcode on Mac OS X 4
- XML (eXtensible Markup Language) 326
- xvalue* (expiring value) **438**
- Y**
 - yield function of namespace `std::this_thread` 819
 - yield the processor 819
 - yield_value function of a coroutine promise object **855**
- Yoda condition 93
- Z**
 - zero-overhead principle of C++ features 470
 - ZIP file format **94**
 - zip_file class from the minix-cpp library 94, 96
 - zip_info class from the minix-cpp library 97