



DEITEL® DEVELOPER SERIES

Modern C++

Functional-Style Programming

Concepts

Templates

Copy/Move Semantics

Spaceship Operator

Smart Pointers

Text Formatting

Security

Modules

Standard Library

C++20 for Programmers

An Objects-Natural Approach

Executors

Design Patterns

Coroutines

Ranges/Views

Performance

Contracts

Open Source Libraries

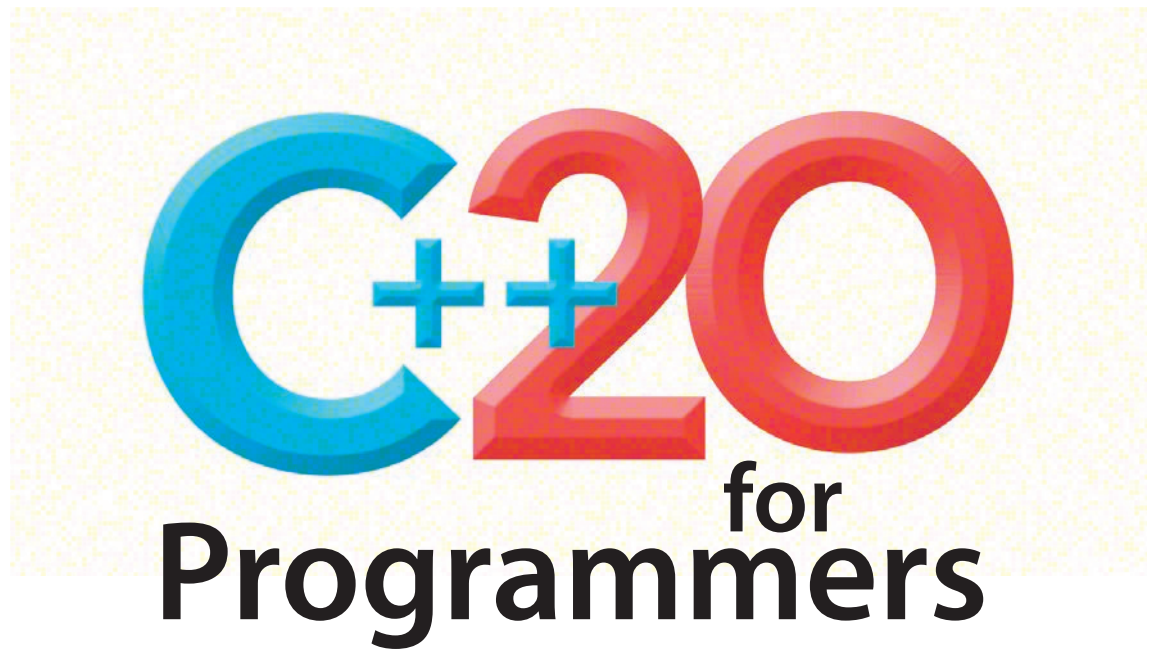
jthread

Parallel Algorithms

Lambdas

Concurrency

PAUL DEITEL • HARVEY DEITEL



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
-------------------------	--------------

I	Intro and Test-Driving Popular, Free C++ Compilers	I
----------	---	----------

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 <code>string</code>	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	Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz	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

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 constexpr	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 vector	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 begin and end	200
7.5.6	Built-In Array Limitations	200
7.6	Using C++20 to_array to Convert a Built-In Array to a std::array	201
7.7	Using const 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	sizeof 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 <i>private</i> and <i>public</i>	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 (<i>::</i>)	287
9.7.7	Member Function <i>setTime</i> and Throwing Exceptions	287
9.7.8	Member Functions <i>to24HourString</i> and <i>to12HourString</i>	288
9.7.9	Implicitly Inlining Member Functions	288
9.7.10	Member Functions vs. Global Functions	288
9.7.11	Using Class <i>Time</i>	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 <i>Time</i>	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 <i>private</i> Data Member	302
9.15	Default Assignment Operator	304
9.16	<i>const</i> Objects and <i>const</i> Member Functions	306
9.17	Composition: Objects as Members of Classes	308
9.18	<i>friend</i> Functions and <i>friend</i> Classes	313
9.19	The <i>this</i> Pointer	314
9.19.1	Implicitly and Explicitly Using the <i>this</i> Pointer to Access an Object's Data Members	315
9.19.2	Using the <i>this</i> Pointer to Enable Cascaded Function Calls	316
9.20	<i>static</i> 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 <i>vector</i> of Objects Containing <i>public</i> Data	327
9.22.2	Serializing a <i>vector</i> of Objects Containing <i>private</i> 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	CommunityMember Class Hierarchy	339
10.2.2	Shape Class Hierarchy and public Inheritance	340
10.3	Relationship Between Base and Derived Classes	341
10.3.1	Creating and Using a SalariedEmployee Class	341
10.3.2	Creating a SalariedEmployee–SalariedCommissionEmployee 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 virtual Functions Are Useful	357
10.7.2	Declaring virtual Functions	357
10.7.3	Invoking a virtual Function	357
10.7.4	virtual Functions in the SalariedEmployee Hierarchy	358
10.7.5	virtual Destructors	361
10.7.6	final Member Functions and Classes	361
10.8	Abstract Classes and Pure virtual Functions	362
10.8.1	Pure virtual 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 Employee	364
10.9.2	Creating Concrete Derived Class SalariedEmployee	367
10.9.3	Creating Concrete Derived Class CommissionEmployee	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— CompensationModel Interface	385
10.12.2	Class Employee	385
10.12.3	CompensationModel Implementations	387
10.12.4	Testing the New Hierarchy	389
10.12.5	Dependency Injection Design Benefits	390
10.13	Runtime Polymorphism with std::variant and std::visit	391
10.14	Multiple Inheritance	397
10.14.1	Diamond Inheritance	401
10.14.2	Eliminating Duplicate Subobjects with virtual Base-Class Inheritance	403
10.15	protected 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	friend 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 DivideByZeroExceptions	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 noexcept	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	assert 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 noexcept(false)	486
12.8	Processing new Failures	487
12.8.1	new Throwing bad_alloc on Failure	488
12.8.2	new Returning nullptr on Failure	489
12.8.3	Handling new Failures Using Function set_new_handler	489
12.9	Standard Library Exception Hierarchy	490
12.10	C++'s Alternative to the finally 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 istream_iterator for Input and ostream_iterator 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	<code>tuple</code> 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

18.4	Creating a Generator Coroutine with <code>co_yield</code> and the generator Library	837
18.5	Launching Tasks with <code>concurrency</code>	841
18.6	Creating a Coroutine with <code>co_await</code> and <code>co_return</code>	845
18.7	Low-Level Coroutines Concepts	853
18.8	C++23 Coroutines Enhancements	855
18.9	Wrap-Up	856

A	Operator Precedence and Grouping	857
----------	---	------------

B	Character Set	859
----------	----------------------	------------

Index	861
--------------	------------

Online Chapters and Appendices

19	Stream I/O and C++20 Text Formatting
20	Other Topics and a Look Toward C++23
C	Number Systems
D	Preprocessor
E	Bit Manipulation

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.

SE 

Appendices

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

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

P.8 How to Get the Online Chapters and Appendices

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

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

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

Online Chapters

- 20 Chapter 19, **Stream I/O; C++20 Text Formatting: A Deeper Look**, discusses standard C++ input/output capabilities and legacy formatting features of the `<iomanip>` library. We include these formatting features primarily for programmers who might encounter them in legacy C++ code. We also present C++20’s new **text-formatting features** in more depth.
- Chapter 20, **Other Topics**, presents miscellaneous C++ topics and looks forward to new features expected in C++23 and beyond.
- 23

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:

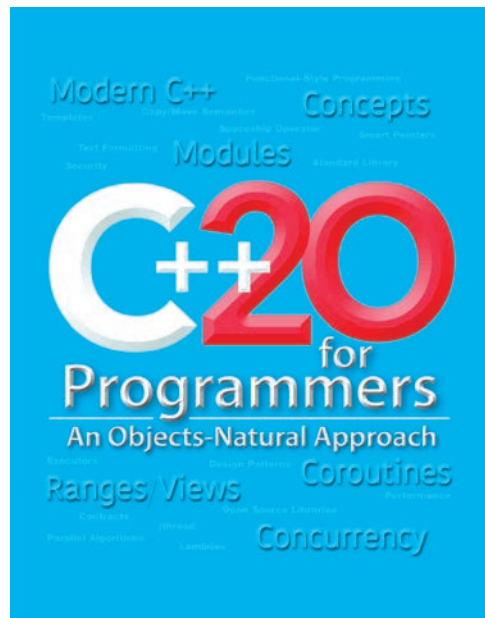
1. Right-click your project name in the **Solution Explorer** and select **Properties**.
2. In the dialog that appears, select **Code Analysis > General** in the left column, then set **Enable Code Analysis on Build** to **Yes** in the right column.
3. Next, select **Code Analysis > Microsoft** in the 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.)

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

Objectives

In this chapter, you'll:

- Quickly get a “40,000-foot view” of this book’s architecture and coverage of the large, complex and powerful programming language that is C++20.
- Test-drive compiling and running a C++ application using our three preferred compilers—Visual C++ in Microsoft Visual Studio on Windows, Clang C++ in Xcode on macOS and GNU g++ on Linux.
- See how to execute Docker containers for the g++ and clang++ command-line compilers, so you can use these compilers on Windows, macOS or Linux.
- See resources where you can learn about C++’s history and milestones over 40+ years.
- Understand why concurrent programming is crucial in Modern C++ for getting maximum performance from today’s multi-core processors.
- Review object-technology concepts used in the early chapters’ objects-natural case studies and presented in the book’s object-oriented programming chapters (starting with Chapter 9).



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

- 1.2.1 Compiling and Running a C++20 Application with Visual Studio 2022 Community Edition on Windows
- 1.2.2 Compiling and Running a C++20 Application with Xcode on macOS
- 1.2.3 Compiling and Running a C++20 Application with GNU C++ on Linux
- 1.2.4 Compiling and Running a C++20 Application with g++ in the GCC Docker Container

- 1.2.5 Compiling and Running a C++20 Application with clang++ in a Docker Container

1.3 Moore’s Law, Multi-Core Processors and Concurrent Programming**1.4** A Brief Refresher on Object Orientation**1.5** Wrap-Up**1.1 Introduction**

Welcome to C++—one of the world’s most popular programming languages.¹ We present Modern C++ in the context of C++20—the latest version standardized through the **International Organization for Standardization (ISO)**. This chapter presents

- a quick way for you to understand the book’s superstructure,
- several test-drives of the most popular free C++ compilers,
- a discussion of Moore’s law, multi-core processors and why concurrent programming is crucial to building high-performance applications in Modern C++, and
- a brief refresher on object-oriented programming concepts and terminology we’ll use throughout the book.

This Book’s Superstructure

Before you “dig in,” we recommend that you get a “40,000-foot” view of the book’s superstructure to understand where you’re headed as you prepare to learn the large, complex and powerful language that is C++20. To do so, we recommend reviewing the following items:

- The one-page, full-color Table of Contents diagram inside the front cover provides a high-level overview of the book. You can view a scalable PDF version of this diagram at
<https://deitel.com/cpp20fpTOCdiagram>
- The back cover contains a concise introduction to the book, a bullet list of its key features and several testimonial comments. More are included on the inside back cover and its facing page. These comments are from the C++ subject-matter experts who reviewed the prepublication manuscript. Reading them will give you a nice overview of the book’s features the reviewers felt were important. These comments are also posted on the book’s webpage at
<https://deitel.com/cpp20fp>
- The **Preface** presents the “soul of the book” and our approach to Modern C++ programming. We introduce the “Objects-Natural Approach,” in which you’ll

1. “TIOBE Index.” Accessed January 10, 2022. <https://www.tiobe.com/tiobe-index/>.

use small numbers of simple C++ statements to make powerful classes perform significant tasks—long before you create custom classes. Be sure to read the Tour of the Book, which points out the key features of each chapter. As you read the Tour, you might also want to refer to the Table of Contents diagram.

Resources on the History of C++

In 1979, Bjarne Stroustrup began creating C++, which he called “C with Classes.”² There are now at least five million developers (with some estimates as high as 7.5 million^{3,4}) using C++ to build a wide range of business-critical and mission-critical systems and applications software.^{5,6} Today’s popular desktop operating systems—Windows⁷ and macOS⁸—are partially written in C++. Many popular applications also are partially written in C++, including web browsers (e.g., Google Chrome⁹ and Mozilla Firefox¹⁰), database management systems (e.g., MySQL¹¹ and MongoDB¹²) and more.

C++’s history and significant milestones are well documented:

- Wikipedia’s C++ page provides a detailed history of C++ with many citations:
<https://en.wikipedia.org/wiki/C%2B%2B>
- Bjarne Stroustrup, C++’s creator, provides a thorough history of the language and its design from inception through C++20:
<https://www.stroustrup.com/C++.html#design>
- [cppreference.com](https://en.cppreference.com/w/cpp/language/history) provides a list of C++ milestones since its inception with many citations:
<https://en.cppreference.com/w/cpp/language/history>

-
2. “Bjarne Stroustrup.” Accessed January 10, 2022. https://en.wikipedia.org/wiki/Bjarne_Stroustrup.
 3. “State of the Developer Nation, 21st Edition,” Q3 2021. Accessed January 10, 2022. <https://www.slashdata.co/free-resources/state-of-the-developer-nation-21st-edition>.
 4. Tim Anderson, “Report: World’s Population of Developers Expands, Javascript Reigns, C# Overtakes PHP,” April 26, 2021. Accessed January 10, 2022. https://www.theregister.com/2021/04/26/report_developers_slashdata/.
 5. “Top 10 Reasons to Learn C++.” Accessed January 10, 2022. <https://www.geeksforgeeks.org/top-10-reasons-to-learn-c-plus-plus/>.
 6. “What Is C++ Used For? Top 12 Real-World Applications and Uses of C++.” Accessed January 10, 2022. <https://www.softwaretestinghelp.com/cpp-applications/>.
 7. “What Programming Language Is Windows Written In?” Accessed January 10, 2022. <https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in>.
 8. “macOS.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. <https://en.wikipedia.org/wiki/MacOS>.
 9. “Google Chrome.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/Google_Chrome.
 10. “Firefox.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. <https://en.wikipedia.org/wiki/Firefox>.
 11. “MySQL.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. <https://en.wikipedia.org/wiki/MySQL>.
 12. “MongoDB.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. <https://en.wikipedia.org/wiki/MongoDB>.

1.2 Test-Driving a C++20 Application

In this section, you'll compile, run and interact with your first C++ application¹³—a guess-the-number game, which picks a random number from 1 to 1,000 and prompts you to guess it. If you guess correctly, the game ends. If you guess incorrectly, the application indicates whether your guess is higher or lower than the correct number. There's no limit on the number of guesses you can make.

Summary of the Compiler and IDE Test-Drives

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

- Microsoft Visual Studio 2022 Community edition for Windows (Section 1.2.1),
- Clang in Apple Xcode on macOS (Section 1.2.2),
- GNU g++ in a shell on Linux (Section 1.2.3),
- g++ in a shell running inside the GNU Compiler Collection (GCC) Docker container (Section 1.2.4), and
- clang++ (the command-line version of the Clang C++ compiler) in a shell running inside a Docker container (Section 1.2.5).

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

1.2.1 Compiling and Running a C++20 Application with Visual Studio 2022 Community Edition on Windows

In this section, you'll run a C++ program on Windows using Microsoft Visual Studio 2022 Community edition.¹⁴ There are several versions of Visual Studio available—on some versions, the options, menus and instructions we present might differ slightly. From this point forward, we'll simply say “Visual Studio” or “the IDE.”

Step 1: Checking Your Setup

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


Step 2: Launching Visual Studio

Open Visual Studio from the **Start** menu. Dismiss this initial Visual Studio window by pressing the *Esc* key. Do not click the **X** in the upper-right corner—that will terminate Visual Studio. You can access this window at any time by selecting **File > Start Window**. We use > to indicate selecting a menu item from a menu, so **File > Open** means “select the **Open** menu item from the **File** menu.”

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

14. At the time of this writing, the Visual Studio 2022 Community version number was 17.0.5.

Step 3: Creating a Project

A **project** is a group of related files, such as the C++ source-code files that compose an application. Visual Studio organizes applications into projects and **solutions**. A solution contains one or more projects. Multi-project solutions are used to create large-scale applications. Each application in this book requires only a single-project solution. For our code examples, you'll begin with an **Empty Project** and add files to it. To create a project: 

1. Select **File > New > Project...** to display the **Create a New Project** dialog.
2. Select the **Empty Project** template with the tags **C++**, **Windows** and **Console**. This project template is for programs that execute at the command line in a Command Prompt window. Depending on your Visual Studio version and its installed options, many other project templates may be installed. You can filter your choices using the **Search for templates** textbox and the drop-down lists below it. Click **Next** to display the **Configure your new project** dialog.
3. Provide a **Project name** and **Location**. For the **Project name**, we specified `cpp20_test`. For the **Location**, we selected this book's `examples` folder. Click **Create** to open your new project in Visual Studio.

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

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

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

When you edit C++ code, Visual Studio displays each file as a separate tab within the window. The **Solution Explorer**—docked to Visual Studio's left or right side—is for viewing and managing your application's files. In this book's examples, you'll typically place each program's code files in the **Source Files** folder. If the **Solution Explorer** is not displayed, you can display it by selecting **View > Solution Explorer**.


Step 4: Adding the `GuessNumber.cpp` File to the Project

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

1. Right-click the **Source Files** folder and select **Add > Existing Item...**
2. In the dialog that appears, navigate to the `ch01` subfolder of the book's `examples` folder, select `GuessNumber.cpp` and click **Add**.¹⁵

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

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

1. Right-click the project's node— `cpp20_test`—in the **Solution Explorer** and select **Properties** to display the project's `cpp20_test` Property Pages dialog.
2. In the **Configuration** drop-down list, select **All Configurations**. In the **Platform** drop-down list, select **All Platforms**.

15. For the multiple source-code-file programs that you'll see in later chapters, select all the files for a given program. When you begin creating programs yourself, you can right-click the **Source Files** folder and select **Add > New Item...** to display a dialog for adding a new file.

3. In the left column, expand the **C/C++** node, then select **Language**.
4. In the right column, click in the field to the right of **C++ Language Standard**, click the down arrow, then select **ISO C++20 Standard (/std:c++20)** and click **OK**.

20

Step 6: Compiling and Running the Project

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

```

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

```

Step 7: Entering Your First Guess

At the `?` prompt, type `500` and press *Enter*—the outputs will vary each time you run the program. In our case, the application displayed "Too low. Try again." to indicate the value was less than the number the application chose as the correct guess:

```

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

```

Step 8: Entering Another Guess

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

```

C:\Users\PaulDeitel\Documents\examples\cpp20_test\x64\Debug\cpp20_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
?

```

Step 9: Entering Additional Guesses

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

```

C:\Users\PaulDette\Documents\examples\cpp20_test\Debug\cpp20_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
? 625
Too high. Try again.
? 562
Too low. Try again.
? 593
Too low. Try again.
? 607
Too low. Try again.
? 616
Too high. Try again.
? 612
Too low. Try again.
? 614
Too high. Try again.
? 613
Excellent! You guessed the number!
Would you like to play again (y or n)?

```

Step 10: Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application.

Reusing This Project for Subsequent Examples

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

Using Ubuntu Linux in the Windows Subsystem for Linux

Some Windows users may want to use the GNU gcc compiler on Windows. You can do this using the **GNU Compiler Collection Docker container** (Section 1.2.4), or you can use gcc in Ubuntu Linux running in the **Windows Subsystem for Linux**. To install the Windows Subsystem for Linux, follow the instructions at

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

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

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

Then you can continue with *Step 2* in Section 1.2.3.

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

In this section, you'll run a C++ program on macOS using Apple's version of the Clang compiler in the Apple Xcode IDE.¹⁶

Step 1: Checking Your Setup

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

Step 2: Launching Xcode

Open a Finder window, select **Applications** and double-click the Xcode icon:



If this is your first time running Xcode, the **Welcome to Xcode** window appears. Close this window—you can access it by selecting **Window > Welcome to Xcode**. We use the > character to indicate selecting a menu item from a menu. For example, **File > Open...** indicates that you should select the **Open...** menu item from the **File** menu.

Step 3: Creating a Project

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

1. Select **File > New > Project...**
2. At the top of the **Choose a template for your new project** dialog, click **macOS**.
3. Under **Application**, click **Command Line Tool** and click **Next**.
4. For **Product Name**, enter a name for your project—we specified `cpp20_test`.
5. In the **Language** drop-down list, select **C++**, then click **Next**.
6. Specify where you want to save your project. We selected the `examples` folder containing this book's code examples.
7. Click **Create**.

Xcode creates your project and displays the **workspace window** initially showing three areas—the **Navigator area** (left), **Editor area** (middle) and **Utilities area** (right).

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

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

Clicking a navigator button displays the corresponding navigator panel.

The middle **Editor** area is for managing project settings and editing source code. This area is always displayed in your workspace window. When you select a file in the **Project** navigator, the file's contents display in the **Editor** area. The right-side **Utilities** area typically displays **inspectors**. For example, if you were building an iPhone app that contained a touchable

16. At the time of this writing, the Xcode version was 13.2.1.

button, you'd be able to configure the button's properties (its label, size, position, etc.) in this area. You will not use the **Utilities** area in this book. There's also a **Debug** area where you'll interact with the running guess-the-number program. This will appear below the **Editor** area.

The workspace window's toolbar contains options for executing a program, displaying the progress of tasks executing in Xcode, and hiding or showing the left (Navigator) and right (Utilities) areas.

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

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

20

1. In the **Project** navigator, select your project's name (cpp20_test).
2. In the **Editors** area's left side, select your project's name under **TARGETS**.
3. At the top of the **Editors** area, click **Build Settings**, and just below it, click **All**.
4. Scroll to the **Apple Clang - Language - C++** section.
5. Click the value to the right of **C++ Language Dialect** and select **GNU++20** [`-std=gnu++20`].
6. Click the value to the right of **C++ Standard Library** and select **Compiler Default**.

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

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

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

In a Finder window, open the `ch01` folder in the book's `examples` folder, then drag `GuessNumber.cpp` onto the `cpp20_test` folder in the **Project** navigator. In the dialog that appears, ensure that **Copy items if needed** is checked, then click **Finish**.¹⁷

Step 7: Compiling and Running the Project

To compile and run the project so you can test-drive the application, simply click the run (▶) button on Xcode's toolbar. If the program compiles correctly, Xcode opens the **Debug** area and executes the program in the right half of the **Debug** area, and the application displays "Please type your first guess." and a question mark (?) as a prompt for input:



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

Step 8: Entering Your First Guess

Click the **Debug** area, then type **500** and press *Return*—the outputs will vary each time you run the program. In our case, the application displayed "Too high. Try again." because the value was more than the number the application chose as the correct guess.

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

Step 9: Entering Another Guess

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

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

Step 10: Entering Additional Guesses

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

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
? 125
Too low. Try again.
? 187
Too high. Try again.
? 156
Too high. Try again.
? 140
Too low. Try again.
? 148
Too high. Try again.
? 144
Too high. Try again.
? 142
Too low. Try again.
? 143

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

Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application.

Reusing This Project for Subsequent Examples

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

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

In this section, you'll run a C++ program in a Linux shell using the GNU C++ compiler (g++).¹⁸ For this test-drive, we assume that you've read the **Before You Begin** section and that you've placed the book's examples in your user account's `Documents/examples` folder.

Step 1: Changing to the ch01 Folder

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

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

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

Step 2: Compiling the Application

Before running the application, you must first compile it with the `g++` command:¹⁹

- The `-std=c++20` option indicates that we're using C++20.

18. At the time of this writing, the current g++ version was 11.2. You can determine your system's g++ version number with the command `g++ --version`. If you have an older version of g++, consider searching online for the instructions to upgrade the GNU Compiler Collection (GCC) for your Linux distribution or consider using the GCC Docker container discussed in Section 1.2.4.

19. If you have multiple g++ versions installed, you might need to use `g++-##`, where ## is the g++ version number. For example, the command `g++-11` might be required to run the latest version of g++ 11.x on your computer.

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

- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program. If you do not include this option, `g++` automatically names the executable `a.out`.

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

Step 3: Running the Application

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

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

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

Step 4: Entering Your First Guess

The application displays "Please type your first guess.", then displays a question mark (?) as a prompt on the next line. At the prompt, enter **500**—the outputs will vary each time you run the program:

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

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

Step 5: Entering Another Guess

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

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

Step 6: Entering Additional Guesses

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

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

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

Step 7: Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application and returns you to the shell.

1.2.4 Compiling and Running a C++20 Application with g++ in the GCC Docker Container

You can use the latest GNU C++ compiler on your system, regardless of your operating system. One of the most convenient cross-platform ways to do this is by using the **GNU Compiler Collection (GCC) Docker container**. This section assumes you've already installed **Docker Desktop** (Windows or macOS) or **Docker Engine** (Linux), as discussed in the **Before You Begin** section that follows the **Preface**.

Executing the GNU Compiler Collection (GCC) Docker Container

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

1. Use `cd` to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the GCC Docker container with the command²⁰

```
docker run --rm -it -v "%CD%":/usr/src gcc:latest
```
3. macOS/Linux users: Launch the GCC Docker container with the command

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

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

In the preceding commands:

- `--rm` cleans up the container's resources when you eventually shut it down.
- `-it` runs the container in interactive mode, so you can enter commands to change folders and to compile and run programs using the GNU C++ compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) allows the Docker container to access the files in the folder from which you executed the `docker run` command. In the Docker container, you'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples. In other words, your local system folder will be mapped to the `/usr/src` folder in the Docker container.
- `gcc:latest` is the container name. The `:latest` specifies that you want to use the most up-to-date version of the `gcc` container.²¹

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

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

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

Changing to the `ch01` Folder in the Docker Container

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

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

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

Terminating the Docker Container

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

1.2.5 Compiling and Running a C++20 Application with `clang++` in a Docker Container

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

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

We used the most recent and widely downloaded one containing `clang++` version 13, which you can get via the following command:²²

```
docker pull teeks99/clang-ubuntu:13
```

Executing the `teeks99/clang-ubuntu` Docker Container

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

1. Use `cd` to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the Docker container with the command²³

```
docker run --rm -it -v "%CD%":/usr/src teeks99/clang-ubuntu:13
```

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

```
docker run --rm -it -v "$(pwd)":/usr/src teeks99/clang-ubuntu:13
```

In the preceding commands:

- `--rm` cleans up the container's resources when you eventually shut it down.
- `-it` runs the container in interactive mode, so you can enter commands to change folders and to compile and run programs using the `clang++` compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) allows the Docker container to access the files in the folder from which you executed the `docker run` command. In the Docker container, you'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples. In other words, your local system folder will be mapped to the `/usr/src` folder in the Docker container.
- `teeks99/clang-ubuntu:13` is the container name.

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

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

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

Changing to the `ch01` Folder in the Docker Container

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

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

22. The version of the Clang C++ compiler used in Xcode is not the most up-to-date version, so it does not have as many C++20 features implemented as the version directly from the LLVM/Clang team. Also, at the time of this writing, using "latest" rather than "13" in the `docker pull` command gives you a Docker container with `clang++` 12, not 13.

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

Compiling the Application

Before running the application, you must first compile it. This container uses the command `clang++-13`, as in

```
clang++-13 -std=c++20 GuessNumber.cpp -o GuessNumber
```

where:

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

Running the Application

To run and interact with the `GuessNumber` application in the Docker container, follow Steps 3–7 of Section 1.2.3’s GNU C++ Test-Drive.

Terminating the Docker Container

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

1.3 Moore’s Law, Multi-Core Processors and Concurrent Programming

Many of today’s personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second. The Japanese Fugaku supercomputer can perform over 442 quadrillion calculations per second (442.01 *petaflops*).²⁴ To put that in perspective, **the Fugaku supercomputer can perform in one second about 40 million calculations for every person on the planet!** And supercomputing “upper limits” are growing quickly.

Perf 

Moore’s Law


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

Perf 

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


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

24. “Top500.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. <https://en.wikipedia.org/wiki/TOP500>.

Key executives at computer-processor companies NVIDIA and Arm have indicated that Moore's law no longer applies.^{25,26} Computer processing power continues to increase but now relies on new processor designs, such as multi-core processors.  Perf

Multi-Core Processors and Performance

Most computers today have **multi-core processors** that economically implement multiple processors on a single integrated circuit chip. A dual-core processor has two CPUs, a quad-core processor has four, and an octa-core processor has eight. Our primary testing computer uses an eight-core Intel processor. Apple's recent M1 Pro and M1 Max processors have 10-core CPUs. In addition, the top-of-the-line M1 Pro has a 16-core GPU, while the top-of-the-line M1 Max processor has a 32-core GPU, and both have a 16-core "neural engine" for machine learning.^{27,28} Intel has some processors with up to 72 cores²⁹ and is working on processors with up to 80.³⁰ AMD is working on processors with 192 and 256 cores.³¹ The number of cores will continue to grow.

In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, thereby enabling your program to complete faster. **To take full advantage of multi-core architecture, you need to write multi-threaded applications.** When a program splits tasks into separate threads, a multi-core system can run those threads in parallel when a sufficient number of cores is available.  Perf

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



1.4 A Brief Refresher on Object Orientation

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in Chapter 9—the **classes** objects come from, are essentially **reusable**

25. "Moore's Law Turns 55: Is It Still Relevant?" Accessed November 2, 2020. <https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant>.
26. "Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow." Accessed November 2, 2020. <https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/>.
27. Juli Clover, "Apple's M1 Pro Chip: Everything You Need to Know," November 3, 2021. Accessed January 19, 2022. <https://www.macrumors.com/guide/m1-pro/>.
28. "Apple M1 Pro and M1 Max." Wikipedia. Wikimedia Foundation. Accessed January 19, 2022. https://en.wikipedia.org/wiki/Apple_M1_Pro_and_M1_Max.
29. "Intel® Xeon Phi™ Processors." Accessed November 28, 2021. <https://ark.intel.com/content/www/us/en/ark/products/series/132784/intel-xeon-phi-72x5-processor-family.html>.
30. Anton Shilov, "Intel's Sapphire Rapids Could Have 72–80 Cores, According to New Die Shots," April 30, 2021. Accessed November 28, 2021. <https://www.tomshardware.com/news/intel-sapphire-rapids-could-feature-80-cores>.
31. Joel Hruska, "Future 256-Core AMD Epyc CPU Might Sport Remarkably Low 600W TDP," November 1, 2021. Accessed November 28, 2021. <https://www.extremetech.com/computing/328692-future-256-core-amd-epyc-cpu-might-sport-remarkably-low-600w-tdp>.

software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any **noun** can be reasonably represented as a software object in terms of **attributes** (e.g., name, color and size) and **behaviors** (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

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

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

Functions, Member Functions and Classes

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

Instantiation

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

Reuse

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

crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

Messages and Member-Function Calls

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

Attributes and Data Members

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

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

Encapsulation

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



Inheritance

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

Object-Oriented Analysis and Design

Soon you’ll be writing programs in C++. How will you create the **code** for your programs? Perhaps, like many programmers, you’ll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the book’s early chapters), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

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

SE 

1.5 Wrap-Up

In this introductory chapter, you saw how to compile and run applications using our three preferred compilers—Visual C++ in Visual Studio 2022 Community edition on Windows, Clang in Xcode on macOS and GNU g++ on Linux. We pointed you to a Microsoft resource for installing Ubuntu Linux using the Windows Subsystem for Linux so you can run g++ on Windows. We also demonstrated how to launch cross-platform Docker containers so you can use the latest g++ and clang++ versions on Windows, macOS or Linux.

We pointed you to several resources for learning about C++'s history and design, including those provided by Bjarne Stroustrup, C++'s creator. Next, we discussed Moore's law, multi-core processors and why Modern C++'s concurrent programming features are crucial for taking advantage of the power multi-core processors provide. Finally, we provided a brief refresher on object-oriented programming concepts and terminology we'll use throughout the book.

In the next chapter, we introduce C++ programming with basic input and output statements, fundamental data types, arithmetic, decision making and our first "Objects Natural" case study on using objects of C++ standard library class `string`.

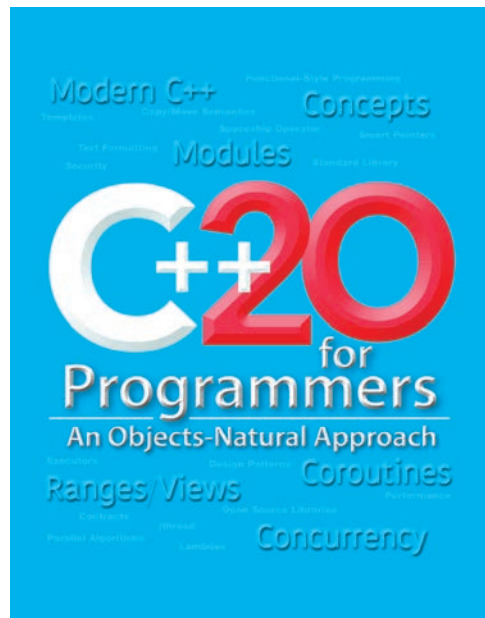
2

Intro to C++20 Programming

Objectives

In this chapter, you'll:

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



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

2.1 Introduction

This chapter presents several code examples that demonstrate how your programs can display messages and obtain data from the user for processing. The first three examples display messages on the screen. The next obtains two numbers from a user at the keyboard, calculates their sum and displays the result—the accompanying discussion introduces C++’s arithmetic operators. The fifth example demonstrates decision making by showing you how to compare two numbers, then display messages based on the comparison results.

The “Objects Natural” Learning Approach

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

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

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

Compiling and Running Programs

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

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

2.2 First Program in C++: Displaying a Line of Text

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

```

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

```

Welcome to C++!

Fig. 2.1 | Text-printing program.

Comments

Lines 1 and 2

```

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

```

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

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

or

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

```

`#include` Preprocessing Directive

Line 3

```
#include <iostream> // enables program to output data to the screen
```

is a **preprocessing directive**—that is, a message to the C++ preprocessor, which the compiler invokes before compiling the program. This line notifies the preprocessor to include in the program the contents of the **input/output stream header** `<iostream>`. This header is a file containing information the compiler requires when compiling any program that outputs data to the screen or inputs data from the keyboard using C++'s stream input/output. The program in Fig. 2.1 outputs data to the screen. Chapter 5 discusses headers in more detail, and online Chapter 19 explains the contents of `<iostream>` in more detail.

Blank Lines and Whitespace

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

The main Function

Line 6

```
int main() {
```

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

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

An Output Statement

Line 7

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

displays the characters contained between the double quotation marks. Together, the quotation marks and the characters between them are called a **string**, a **character string** or a **string literal**. We refer to characters between double quotation marks simply as strings. Whitespace characters in strings are not ignored by the compiler.

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



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

Indentation

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

The std Namespace

The `std::` before `cout` is required when we use names that we've brought into the program from standard-library headers like `<iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to **namespace std**.¹ The names `cin` (the standard input stream) and `cerr` (the standard error stream)—introduced in Chapter 1—also belong to namespace `std`. We discuss namespaces in Chapter 16. For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and

1. We pronounce “`std::`” as “standard,” rather as its individual letters `s`, `t` and `d`.

cerr in a program. This can be cumbersome—we’ll soon introduce using declarations and the using directive, which will enable you to omit `std::` before each use of a name in the `std` namespace.

The Stream Insertion Operator and Escape Sequences

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

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

The return Statement

Line 9

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

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

2.3 Modifying Our First C++ Program

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

Displaying a Single Line of Text with Multiple Statements

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

```

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

```

```

Welcome to C++!

```

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

Displaying Multiple Lines of Text with a Single Statement

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

```

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

```

```

Welcome
to

C++!

```

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

2.4 Another C++ Program: Adding Integers

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

```

1 // fig02_04.cpp
2 // Addition program that displays the sum of two integers.
3 #include <iostream> // enables program to perform input and output
4

```

Fig. 2.4 | Addition program that displays the sum of two integers. (Part I of 2.)

```

5 // function main begins program execution
6 int main() {
7     // declaring and initializing variables
8     int number1{0}; // first integer to add (initialized to 0)
9     int number2{0}; // second integer to add (initialized to 0)
10    int sum{0}; // sum of number1 and number2 (initialized to 0)
11
12    std::cout << "Enter first integer: "; // prompt user for data
13    std::cin >> number1; // read first integer from user into number1
14
15    std::cout << "Enter second integer: "; // prompt user for data
16    std::cin >> number2; // read second integer from user into number2
17
18    sum = number1 + number2; // add the numbers; store result in sum
19
20    std::cout << "Sum is " << sum << "\n"; // display sum
21 } // end function main

```

```

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

```

Fig. 2.4 | Addition program that displays the sum of two integers. (Part 2 of 2.)

Variable Declarations and Braced Initialization

Lines 8–10

```

int number1{0}; // first integer to add (initialized to 0)
int number2{0}; // second integer to add (initialized to 0)
int sum{0}; // sum of number1 and number2 (initialized to 0)

```

are **declarations**—`number1`, `number2` and `sum` are the names of **variables**. These declarations specify that the variables `number1`, `number2` and `sum` are data of type **int**, meaning they will hold **integer** (whole number) values, such as 7, −11, 0 and 31914. All variables must be declared with a name and a data type.

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

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

```

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

```

In legacy C++ programs, you're likely to encounter initialization statements using this older C++ coding style. In subsequent chapters, we'll discuss various benefits of braced initializers.

Declaring Multiple Variables at Once

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

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

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

Fundamental Types

We'll soon discuss the type `double` for specifying real numbers and the type `char` for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and -11.19. A `char` variable may hold only a single lowercase letter, uppercase letter, digit or special character (e.g., \$ or *). Types such as `int`, `double`, `char` and `long long` are called **fundamental types**. Fundamental-type names typically consist of one or more keywords and must appear in all lowercase letters. For a complete list of C++ fundamental types and their typical ranges, see

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

Identifiers and Camel-Case Naming

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

C++ allows identifiers of any length. Do not begin an identifier with an underscore and a capital letter or two underscores—C++ compilers use names like that for their own purposes internally.

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter—e.g., `firstNumber` starts its second word, `Number`, with a capital N. This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps.

Placement of Variable Declarations

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

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

could have been placed immediately before line 13:

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

the declaration in line 9:

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

could have been placed immediately before line 16:

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

and the declaration in line 10:

```
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

could have been placed immediately before line 18:

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

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

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

Obtaining the First Value from the User

Line 12

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

displays `Enter first integer:` followed by a space. This message is called a **prompt** because it directs the user to take a specific action. Line 13

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

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

When the preceding statement executes, the program waits for you to enter a value for variable `number1`. You respond by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the program. The `cin` object converts the character representation of the number to an integer value and assigns this value to the variable `number1`. Pressing *Enter* also causes the cursor to move to the beginning of the next line on the screen.

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

Obtaining the Second Value from the User

Line 15

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

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

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

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

Calculating the Sum of the Values Input by the User

The assignment statement in line 18

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

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

Displaying the Result

Line 20

```
std::cout << "Sum is " << sum << "\n"; // display sum
```

displays the character string `"Sum is "` followed by the numerical value of variable `sum` and a newline.

The preceding statement outputs multiple values of different types. The stream insertion operator “knows” how to output each type of data. Using multiple stream insertion

operators (<<) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**.

Calculations can also be performed in output statements. We could have eliminated the variable `sum` by combining the statements in lines 18 and 20 into the statement

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

The signature feature of C++ is that you can create your own data types called classes (we discuss this topic beginning in Chapter 9). You can then “teach” C++ how to input and output values of these new data types using the >> and << operators, respectively. This is called **operator overloading**, which we explore in Chapter 11.

2.5 Arithmetic

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

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

Note the use of various special symbols not used in algebra. The **asterisk** (*) indicates multiplication and the **percent sign** (%) is the remainder operator, which we’ll discuss shortly. These arithmetic operators are all binary operators.

Integer Division

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

Remainder Operator

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

Parentheses for Grouping Subexpressions

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

Rules of Operator Precedence

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

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

$$(a * (b + c))$$

expressions in the innermost pair of parentheses evaluate first.

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

Online Appendix A contains the complete operator precedence chart. **Caution:** In an expression such as $(a + b) * (c - d)$, where two sets of parentheses are not nested but appear “on the same level,” the C++ Standard does not specify the order in which these parenthesized subexpressions will evaluate.

Operator Grouping

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

$$a + b + c$$

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

2.6 Decision Making: Equality and Relational Operators

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

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

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

Reversing the order of the pair of symbols in the operators `!=`, `>=` and `<=` (by writing them as `!=`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `!=` as `=!` will not be a syntax error, but almost certainly it will be a logic error that has an effect at execution time. You'll understand why when we cover logical operators in Section 4.11.

Confusing `==` and `=`

Confusing the equality operator `==` with the assignment operator `=` results in logic errors. We like to read the equality operator as “is equal to” or “double equals” and the assignment operator as “gets” or “gets the value of” or “is assigned the value of.” Confusing these operators may not necessarily cause an easy-to-recognize syntax error, but it may cause subtle logic errors. Compilers generally warn about this.

Using the `if` Statement

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

```

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

```

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

```

28
29     if (number1 > number2) {
30         cout << number1 << " > " << number2 << "\n";
31     }
32
33     if (number1 <= number2) {
34         cout << number1 << " <= " << number2 << "\n";
35     }
36
37     if (number1 >= number2) {
38         cout << number1 << " >= " << number2 << "\n";
39     }
40 } // end function main

```

```

Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7

```

```

Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12

```

```

Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7

```

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

using Declarations

Lines 6–7

```

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

```

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

using Directive

In place of lines 6–7, many programmers prefer the **using directive**

```

using namespace std;

```

which enables your program to use names from the `std` namespace without the `std::` qualification. In the early chapters, we'll use this directive in our programs to simplify the code.²

2. In online Chapter 19, we'll discuss some disadvantages of using directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User

Lines 11–12

```
int number1{0}; // first integer to compare (initialized to 0)
int number2{0}; // second integer to compare (initialized to 0)
```

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

Line 15

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

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

Comparing Numbers

The `if` statement in lines 17–19

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

determines whether the values of variables `number1` and `number2` are equal. If so, the `cout` statement displays a line of text indicating that the numbers are equal. For each condition that is true in the remaining `if` statements starting in lines 21, 25, 29, 33 and 37, the corresponding `cout` statement displays an appropriate line of text.

Braces and Blocks

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

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



Common Logic Error: Placing a Semicolon after a Condition

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

Splitting Lengthy Statements

A lengthy statement may be spread over several lines. If you must do this, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, it's a good practice to indent all subsequent lines.

Operator Precedence and Grouping

With the exception of the assignment operator `=`, all the operators presented in this chapter group from left-to-right. Assignments (`=`) group from right-to-left. So, an expression such as `x = y = 0` evaluates as if it had been written `x = (y = 0)`, which first assigns 0 to `y`, then assigns the result of that assignment (that is, 0) to `x`.

Refer to the complete operator-precedence chart in online Appendix A when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression.

2.7 Objects Natural: Creating and Using Objects of Standard-Library Class `string`

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

You've already worked with C++ objects—specifically the `cout` and `cin` objects, which encapsulate the mechanisms for output and input, respectively. These objects were created for you behind the scenes using classes from the header `<iostream>`. In this section, you'll create and interact with objects of the C++ standard library's `string`³ class.

Test-Driving Class `string`

Classes cannot execute by themselves. A `Person` object can drive a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car's internal mechanisms work. Similarly, the `main` function can “drive” a `string` object by calling its member functions—without knowing how the class is implemented. In this sense, `main` in the following program is referred to as a **driver program**. Figure 2.6's `main` function test-drives several `string` member functions.

```

1 // fig02_06.cpp
2 // Standard library string class test program.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main() {
8     string s1{"happy"};
9     string s2{" birthday"};
10    string s3; // creates an empty string

```

Fig. 2.6 | Standard library `string` class test program. (Part 1 of 2.)

3. You'll learn additional `string` capabilities in subsequent chapters. Chapter 8 discusses class `string` in detail, test-driving many more of its member functions.

```

11
12 // display the strings and show their lengths
13 cout << "s1: \"\" << s1 << "\"; length: \" << s1.length()
14     << "\"s2: \"\" << s2 << "\"; length: \" << s2.length()
15     << "\"s3: \"\" << s3 << "\"; length: \" << s3.length();
16
17 // compare strings with == and !=
18 cout << "\n\nThe results of comparing s2 and s1:" << boolalpha
19     << "\"s2 == s1: \" << (s2 == s1)
20     << "\"s2 != s1: \" << (s2 != s1);
21
22 // test string member function empty
23 cout << "\n\nTesting s3.empty():\n";
24
25 if (s3.empty()) {
26     cout << "s3 is empty; assigning to s3;\n";
27     s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2
28     cout << "s3: \"\" << s3 << "\"";
29 }
30
31 // testing new C++20 string member functions
32 cout << "\n\ns1 starts with \"ha\": \" << s1.starts_with("ha") << "\n";
33 cout << "s2 starts with \"ha\": \" << s2.starts_with("ha") << "\n";
34 cout << "s1 ends with \"ay\": \" << s1.ends_with("ay") << "\n";
35 cout << "s2 ends with \"ay\": \" << s2.ends_with("ay") << "\n";
36 }

```

```

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

The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true

Testing s3.empty():
s3 is empty; assigning to s3;
s3: "happy birthday"

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

```

Fig. 2.6 | Standard library string class test program. (Part 2 of 2.)

Instantiating Objects

Typically, you cannot call a member function of a class until you create an object of that class⁴—also called instantiating an object. Lines 8–10 create three `string` objects:

- `s1` is initialized with a copy of the string literal `"happy"`,

4. You'll see in Section 9.20 that you can call a class's static member functions without creating an object of that class.

- `s2` is initialized with a copy of the string literal `" birthday"`, and
- `s3` is initialized by default to the **empty `string`** (that is, `""`).

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

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

string Member Function `length`

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

```
s1.length()
```

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

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

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

Comparing string Objects with the Equality Operators

Like numbers, strings can be compared with one another. Lines 18–20 compare `s2` to `s1` using the equality operators—string comparisons are case sensitive.⁵

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

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

string Member Function empty

Line 25 calls string member function `empty`, which returns `true` if the `string` is empty—that is, the length of the `string` is 0. Otherwise, `empty` returns `false`. The object `s3` was initialized by default to the empty string, so it is indeed empty, and the body of the `if` statement will execute.

string Concatenation and Assignment

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

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

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

2.8 Wrap-Up

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

Finally, we introduced our “Objects Natural” approach to learning C++ by creating objects of the C++ standard-library class `string` and interacting with them using equality operators and `string` member functions. In subsequent chapters, you’ll create and use many objects of existing classes to accomplish significant tasks with minimal amounts of code. Then, in Chapters 9–11, you’ll create your own custom classes. You’ll see that C++ enables you to “craft valuable classes.” In the next chapter, we begin our introduction to control statements, which specify the order in which a program’s actions are performed.

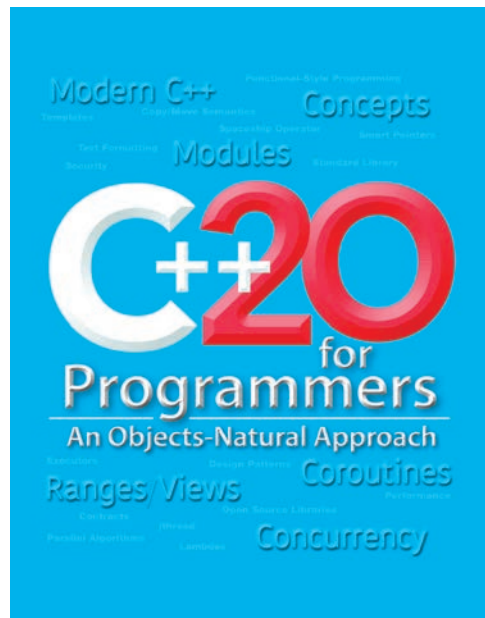
3

Control Statements: Part I

Objectives

In this chapter, you'll:

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



Outline

3.1 Introduction	3.7.2 Converting Between Fundamental Types Explicitly and Implicitly
3.2 Control Structures	3.7.3 Formatting Floating-Point Numbers
3.2.1 Sequence Structure	3.8 Nested Control Statements
3.2.2 Selection Statements	3.8.1 Problem Statement
3.2.3 Iteration Statements	3.8.2 Implementing the Program
3.2.4 Summary of Control Statements	3.8.3 Preventing Narrowing Conversions with Braced Initialization
3.3 <code>if</code> Single-Selection Statement	3.9 Compound Assignment Operators
3.4 <code>if...else</code> Double-Selection Statement	3.10 Increment and Decrement Operators
3.4.1 Nested <code>if...else</code> Statements	3.11 Fundamental Types Are Not Portable
3.4.2 Blocks	3.12 Objects Natural Case Study: Arbitrary-Sized Integers
3.4.3 Conditional Operator (<code>?:</code>)	3.13 C++20: Text Formatting with Function <code>format</code>
3.5 <code>while</code> Iteration Statement	3.14 Wrap-Up
3.6 Counter-Controlled Iteration	
3.6.1 Implementing Counter-Controlled Iteration	
3.6.2 Integer Division and Truncation	
3.7 Sentinel-Controlled Iteration	
3.7.1 Implementing Sentinel-Controlled Iteration	

3.1 Introduction

In this chapter and the next, we present the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects. We discuss the `if` statement in additional detail and introduce the `if...else` and `while` statements. We also introduce the compound assignment operators and the increment and decrement operators.

We discuss why the fundamental types are not portable. We continue our Objects Natural approach with a case study on arbitrary-sized integers that can represent values beyond the ranges of integers supported by computer hardware.

20 We begin introducing C++20's new text-formatting capabilities, which are based on those in Python, Microsoft's .NET languages (like C# and Visual Basic) and Rust.¹ The C++20 capabilities are more concise and more powerful than those in earlier C++ versions.

3.2 Control Structures

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of many problems experienced by software development groups. The blame was pointed at the **goto statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program.

The research of Böhm and Jacopini² had demonstrated that programs could be written without any `goto` statements. The challenge for programmers of the era was to shift their styles to “goto-less programming.” The term **structured programming** became

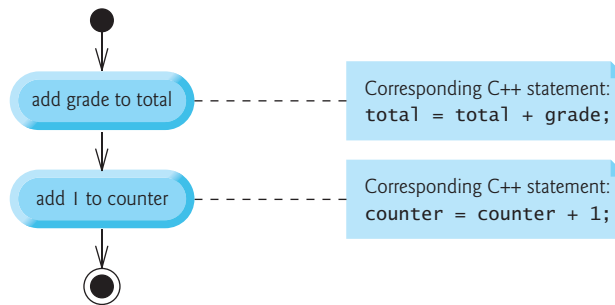
1. Victor Zverovich, “Text Formatting,” July 16, 2019. Accessed November 11, 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html>.
2. C. Böhm and G. Jacopini, “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules,” *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

almost synonymous with “goto elimination.” The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug-free in the first place.

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

3.2.1 Sequence Structure

The sequence structure is built into C++. Unless directed otherwise, statements execute one after the other in the order they appear in the program—that is, in sequence. The following UML³ **activity diagram** illustrates a typical sequence structure in which two calculations are performed in order:



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

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

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

The **solid circle** at the top of the activity diagram represents the **initial state**—the beginning of the workflow before the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram represents the **final state**—that is, the end of the workflow after the program performs its actions.

The sequence-structure activity diagram also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in C++)—explanatory remarks

3. We use the UML in this chapter and Chapter 4 to show the flow of control in control statements, then use UML again in Chapters 9–10 when we present custom class development.

that describe the purpose of symbols in the diagram. A **dotted line** connects each note with the element it describes. This diagram's UML notes show how the diagram relates to the C++ code for each action state. Activity diagrams usually do not show the C++ code.

3.2.2 Selection Statements

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

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

3.2.3 Iteration Statements

C++ provides four **iteration statements**—also called **repetition statements** or **looping statements**—for performing statements repeatedly while a **loop-continuation condition** remains true. The iteration statements are the `while`, `do...while`, `for` and range-based `for`. The `while` and `for` statements perform their action (or group of actions) zero or more times. If the loop-continuation condition is initially false, the action (or group of actions) does not execute. The `do...while` statement performs its action (or group of actions) one or more times. Chapter 4 presents the `do...while` and `for` statements. Chapter 6 presents the range-based `for` statement.

Keywords

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

C++ keywords				
<code>alignas</code>	<code>alignof</code>	<code>and</code>	<code>and_eq</code>	<code>asm</code>
<code>auto</code>	<code>bitand</code>	<code>bitor</code>	<code>bool</code>	<code>break</code>
<code>case</code>	<code>catch</code>	<code>char</code>	<code>char16_t</code>	<code>char32_t</code>
<code>class</code>	<code>compl</code>	<code>const</code>	<code>const_cast</code>	<code>constexpr</code>
<code>continue</code>	<code>decltype</code>	<code>default</code>	<code>delete</code>	<code>do</code>
<code>double</code>	<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>	<code>explicit</code>
<code>export</code>	<code>extern</code>	<code>false</code>	<code>final</code>	<code>float</code>
<code>for</code>	<code>friend</code>	<code>goto</code>	<code>if</code>	<code>import</code>
<code>inline</code>	<code>int</code>	<code>long</code>	<code>module</code>	<code>mutable</code>

C++ keywords (Cont.)

namespace	new	noexcept	not	not_eq
nullptr	operator	or	or_eq	override
private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned
using	void	volatile	virtual	wchar_t
while	xor	xor_eq		
Keywords new in C++20				
char8_t	concept	constexpr	constinit	co_await
co_return	co_yield	requires		

20

3.2.4 Summary of Control Statements

C++ has only three kinds of control structures, which from this point forward, we refer to as control statements:

- sequence,
- selection (if, if...else and switch) and
- iteration (while, do...while, for and range-based for).

You form every program by combining these statements as appropriate for the algorithm you're implementing. We can model each control statement as an activity diagram. Each diagram contains an initial state and a final state representing a control statement's entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build readable programs—we simply connect the exit point of one to the entry point of the next using **control-statement stacking**. There's only one other way in which you may connect control statements—**control-statement nesting**, in which one control statement appears inside another. Thus, algorithms in C++ programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.

3.3 if Single-Selection Statement

We introduced the if single-selection statement briefly in Section 2.6. Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The following C++ statement determines whether the condition `studentGrade >= 60` is true:

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

If so, "Passed" is printed, and the next statement in order is performed. If the condition is false, the output statement is ignored, and the next statement in order is performed. The

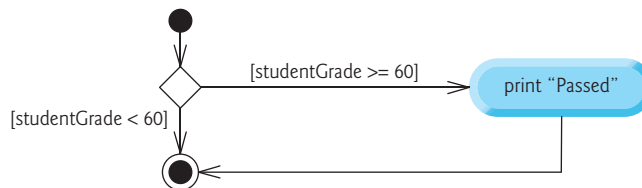
indentation of the second line of this selection statement is optional but recommended for program clarity.

bool Data Type

In Chapter 2, you created conditions using the relational or equality operators. Actually, any expression that evaluates to zero or nonzero can be used as a condition. Zero is treated as false, and nonzero is treated as true. C++ also provides the data type **bool** for Boolean variables that can hold only the values **true** and **false**—each is a C++ keyword. The compiler can implicitly convert **true** to 1 and **false** to 0.

UML Activity Diagram for an if Statement

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



This figure contains the most important symbol in an activity diagram—the diamond, or **decision symbol**, which indicates that a decision is to be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets next to the arrow). If a guard condition is true, the workflow enters the action state to which the transition arrow points. The diagram shows that if the grade is greater than or equal to 60 (i.e., the condition is true), the program prints "Passed" then transitions to the activity's final state. If the grade is less than 60 (i.e., the condition is false), the program immediately transitions to the final state without displaying a message. The **if** statement is a single-entry/single-exit control statement.

3.4 if...else Double-Selection Statement

The **if** single-selection statement performs an indicated action only when the condition is true. The **if...else double-selection statement** allows you to specify an action to perform when the condition is true and another action when the condition is false. For example, the following C++ statement prints "Passed" if `studentGrade >= 60`, but prints "Failed" if it's less than 60:

```

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

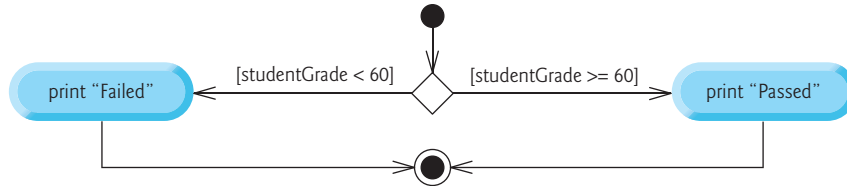
```

In either case, after printing occurs, the next statement in sequence is performed.

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

UML Activity Diagram for an if...else Statement

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



3.4.1 Nested if...else Statements

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

```

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

```

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

```

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

```

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

3.4.2 Blocks

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

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

```

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

```

If `studentGrade` is less than 60, the program executes both statements in the body of the `else` and prints

```

Failed
You must retake this course.

```

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

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

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

Empty Statement

Just as a block can be placed anywhere a single statement can be placed, it’s also possible to have an **empty statement**, which is simply a semicolon (`;`) where a statement typically would be. An empty statement has no effect.

3.4.3 Conditional Operator (?:)

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

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

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

```
studentGrade >= 60
```

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

3.5 while Iteration Statement

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

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

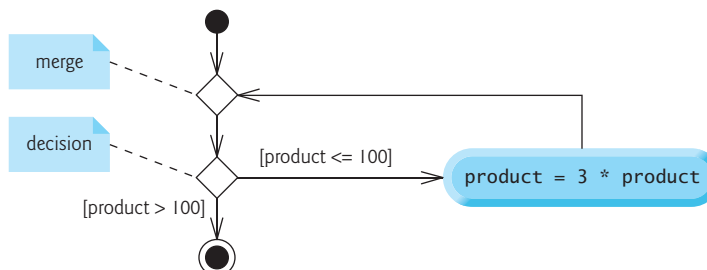
```
int product{3};

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

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

UML Activity Diagram for a while Statement

The following `while` statement UML activity diagram introduces the **merge symbol**:



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

You can distinguish the decision and merge symbols by the number of incoming and outgoing transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that decision. Also, each arrow pointing out of a decision symbol has a guard condition. A merge symbol has two or more transition arrows pointing to it and only one pointing from it to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

3.6 Counter-Controlled Iteration

Consider the following problem statement:

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

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

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

3.6.1 Implementing Counter-Controlled Iteration

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

```

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

```

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

```

19
20 // termination phase
21 int average{total / 10}; // int division yields int result
22
23 // display total and average of grades
24 cout << "\nTotal of all 10 grades is " << total;
25 cout << "\nClass average is " << average << "\n";
26 }

```

```

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

Total of all 10 grades is 846
Class average is 84

```

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

Local Variables in `main`

Lines 8, 9, 14 and 21 declare `int` variables `total`, `gradeCounter`, `grade` and `average`, respectively. Variable `grade` stores the user input. A variable declared in a block (such as a function's body) is a local variable that can be used only from the line of its declaration to the closing right brace of the block. A local variable's declaration must appear before the variable is used. Variable `grade`—declared in the body of the `while` loop—can be used only in that block.

Initializing Variables `total` and `gradeCounter`

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

Reading 10 Grades from the User

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

Calculating and Displaying the Class Average

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

3.6.2 Integer Division and Truncation

This example's average calculation produces an `int` result. The program's sample execution shows that the sum of the grades is 846—when divided by 10, this should yield 84.6. Numbers like 84.6 containing decimal points are **floating-point numbers**. However, in the class-average program, `total / 10` produces the integer 84 because `total` and 10 are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is truncated. The next section shows how to obtain a floating-point result from the averaging calculation. For example, $7 / 4$ yields 1.75 in conventional arithmetic but truncates to 1 in integer arithmetic rather than rounding to 2.

3.7 Sentinel-Controlled Iteration

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

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

In the previous class-average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. Here, we do not know how many grades the user will enter during the program's execution. The program must process an *arbitrary* number of grades.

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

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

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

3.7.1 Implementing Sentinel-Controlled Iteration

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

```

1 // fig03_02.cpp
2 // Solving the class-average problem using sentinel-controlled iteration.
3 #include <iostream>
4 #include <iomanip> // parameterized stream manipulators
5 using namespace std;
6
7 int main() {
8     // initialization phase
9     int total{0}; // initialize sum of grades
10    int gradeCounter{0}; // initialize # of grades entered so far
11
12    // processing phase
13    // prompt for input and read grade from user
14    cout << "Enter grade or -1 to quit: ";
15    int grade;
16    cin >> grade;
17
18    // loop until sentinel value is read from user
19    while (grade != -1) {
20        total = total + grade; // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22
23        // prompt for input and read next grade from user
24        cout << "Enter grade or -1 to quit: ";
25        cin >> grade;
26    }
27
28    // termination phase
29    // if user entered at least one grade...
30    if (gradeCounter != 0) {
31        // use number with decimal point to calculate average of grades
32        double average{static_cast<double>(total) / gradeCounter};
33
34        // display total and average (with two digits of precision)
35        cout << "\nTotal of the " << gradeCounter
36             << " grades entered is " << total;
37        cout << setprecision(2) << fixed;
38        cout << "\nClass average is " << average << "\n";
39    }
40    else { // no grades were entered, so output appropriate message
41        cout << "No grades were entered\n";
42    }
43 }

```

```

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

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

```

Fig. 3.2 | Solving the class-average problem using sentinel-controlled iteration.

Recall that integer division produces an integer result. This program introduces a **cast operator** to force the average calculation to produce a floating-point result. This program also stacks control statements in sequence—the `while` statement is followed in sequence by an `if...else` statement. Much of this program's code is identical to Fig. 3.1, so we concentrate on only the new concepts.

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

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

Compare the program logic for sentinel-controlled iteration in this program with that for counter-controlled iteration in Fig. 3.1. In counter-controlled iteration, each iteration of the `while` statement (lines 12–18 of Fig. 3.1) reads a value from the user for the specified number of iterations. In sentinel-controlled iteration, the program prompts for and reads the first value (lines 14 and 16 of Fig. 3.2) before reaching the `while`. This value determines whether the flow of control should enter the `while`'s body. If the condition is false, the user entered the sentinel value, so no grades were entered, and the body does not execute. If the condition is true, the body begins execution, and the loop adds the grade value to the `total` and increments the `gradeCounter`. Then lines 24–25 in the loop body input the next value from the user. Next, program control reaches the closing right brace of the loop at line 26, so execution continues with the test of the `while`'s condition (line 19). The condition uses the most recent grade entered by the user to determine whether the loop body should execute again.

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

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

3.7.2 Converting Between Fundamental Types Explicitly and Implicitly

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

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

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

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

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

`static_cast` Operator

To perform a floating-point calculation with integers in this example, you first create temporary floating-point values using the **static_cast operator**. Line 32 converts a tempo-

rary copy of its operand in parentheses (`total`) to the type in angle brackets (`double`). The value stored in the `int` variable `total` is still an integer. Using a cast operator in this manner is called **explicit conversion**. `static_cast` is one of several cast operators we'll discuss.

Promotions

After the cast, the calculation consists of the temporary `double` copy of `total` divided by the `int` `gradeCounter`. For arithmetic, the compiler knows how to evaluate only expressions in which all the operand types are identical. To ensure this, the compiler performs an operation called **promotion** (also called **implicit conversion**) on selected operands. In an expression containing values of data types `int` and `double`, C++ **promotes** `int` operands to `double` values. So in line 32, C++ promotes a temporary copy of `gradeCounter`'s value to type `double`, then performs the division. Finally, `average` is initialized with the floating-point result. Section 5.6 discusses the allowed fundamental-type promotions.

Cast Operators for Any Type

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

3.7.3 Formatting Floating-Point Numbers

Figure 3.2's formatting features are introduced here briefly. Online Chapter 19 explains them in depth.

setprecision Parameterized Stream Manipulator

Line 37's call to **setprecision**—`setprecision(2)`—indicates that floating-point values should be output with *two* digits of **precision** to the right of the decimal point (e.g., 92.37). `setprecision` is a **parameterized stream manipulator** because it requires an argument (in this case, 2) to perform its task. Programs that use parameterized stream manipulators must include the header `<iomanip>` (line 4).

fixed Nonparameterized Stream Manipulator

The **non-parameterized stream manipulator** **fixed** (line 37) does not require an argument and indicates that floating-point values should be output in **fixed-point format**. This is as opposed to **scientific notation**⁴, which displays a number between 1.0 and 10.0, multiplied by a power of 10. So, in scientific notation, the value 3,100.0 is displayed as 3.1e+03 (that is, 3.1×10^3). This format is useful for displaying very large or very small values.

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

4. Formatting using scientific notation is discussed further in online Chapter 19.