

FIFTH EDITION



Professional

C++

Marc Gregoire

PROFESSIONAL C++

INTRODUCTION	xlvii	
▶ PART I	INTRODUCTION TO PROFESSIONAL C++	
CHAPTER 1	A Crash Course in C++ and the Standard Library..... 3	
CHAPTER 2	Working with Strings and String Views	87
CHAPTER 3	Coding with Style	111
▶ PART II	PROFESSIONAL C++ SOFTWARE DESIGN	
CHAPTER 4	Designing Professional C++ Programs	137
CHAPTER 5	Designing with Objects	169
CHAPTER 6	Designing for Reuse.....	187
▶ PART III	C++ CODING THE PROFESSIONAL WAY	
CHAPTER 7	Memory Management	211
CHAPTER 8	Gaining Proficiency with Classes and Objects.....	249
CHAPTER 9	Mastering Classes and Objects	283
CHAPTER 10	Discovering Inheritance Techniques.....	337
CHAPTER 11	Odds and Ends.....	397
CHAPTER 12	Writing Generic Code with Templates.....	421
CHAPTER 13	Demystifying C++ I/O	465
CHAPTER 14	Handling Errors.....	495
CHAPTER 15	Overloading C++ Operators	535
CHAPTER 16	Overview of the C++ Standard Library	573
CHAPTER 17	Understanding Iterators and the Ranges Library.....	603
CHAPTER 18	Standard Library Containers	627
CHAPTER 19	Function Pointers, Function Objects, and Lambda Expressions.....	699
CHAPTER 20	Mastering Standard Library Algorithms.....	725

Continues

CHAPTER 21	String Localization and Regular Expressions	763
CHAPTER 22	Date and Time Utilities	793
CHAPTER 23	Random Number Facilities	809
CHAPTER 24	Additional Library Utilities	821
▶ PART IV	MASTERING ADVANCED FEATURES OF C++	
CHAPTER 25	Customizing and Extending the Standard Library	833
CHAPTER 26	Advanced Templates	877
CHAPTER 27	Multithreaded Programming with C++	915
▶ PART V	C++ SOFTWARE ENGINEERING	
CHAPTER 28	Maximizing Software Engineering Methods	971
CHAPTER 29	Writing Efficient C++	993
CHAPTER 30	Becoming Adept at Testing	1021
CHAPTER 31	Conquering Debugging	1045
CHAPTER 32	Incorporating Design Techniques and Frameworks	1083
CHAPTER 33	Applying Design Patterns	1105
CHAPTER 34	Developing Cross-Platform and Cross-Language Applications . . .	1137
▶ PART VI	APPENDICES	
APPENDIX A	C++ Interviews	1165
APPENDIX B	Annotated Bibliography	1191
APPENDIX C	Standard Library Header Files	1203
APPENDIX D	Introduction to UML	1213
INDEX	1219

PROFESSIONAL

C++

PROFESSIONAL
C++

Fifth Edition

Marc Gregoire



Professional C++

Copyright © 2021 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada and the United Kingdom

ISBN: 978-1-119-69540-0

ISBN: 978-1-119-69550-9 (ebk)

ISBN: 978-1-119-69545-5 (ebk)

Manufactured in the United States of America

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2020950208

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

*Dedicated to my wonderful parents and my brother,
who are always there for me. Their support and
patience helped me in finishing this book.*

ABOUT THE AUTHOR

MARC GREGOIRE is a software architect from Belgium. He graduated from the University of Leuven, Belgium, with a degree in “Burgerlijk ingenieur in de computer wetenschappen” (equivalent to a master of science in engineering in computer science). The year after, he received an advanced master’s degree in artificial intelligence, *cum laude*, at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working in international teams stretching from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is a software architect at Nikon Metrology (nikonmetrology.com), a division of Nikon and a leading provider of precision optical instruments, X-ray machines, and metrology solutions for X-ray, CT, and 3-D geometric inspection.

His main expertise is C/C++, specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24/7 on Windows and Linux platforms: for example, KNX/EIB home automation software. In addition to C/C++, Marc also likes C#.

Since April 2007, he has received the annual Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (becpp.org), co-author of *C++ Standard Library Quick Reference* 1st and 2nd editions (Apress), a technical editor for numerous books for several publishers, and a regular speaker at the CppCon C++ conference. He maintains a blog at www.nuonsoft.com/blog/ and is passionate about traveling and gastronomic restaurants.

ABOUT THE TECHNICAL EDITORS

PETER VAN WEERT is a Belgian software engineer whose main interests and expertise are application software development, programming languages, algorithms, and data structures.

He received his master of science degree in computer science *summa cum laude* with congratulations from the Board of Examiners from the University of Leuven. In 2010, he completed his PhD thesis on the design and efficient compilation of rule-based programming languages at the research group for declarative programming languages and artificial intelligence. During his doctoral studies he was a teaching assistant for object-oriented programming (Java), software analysis and design, and declarative programming.

Peter then joined Nikon Metrology, where he worked on large-scale, industrial application software in the area of 3-D laser scanning and point cloud inspection for over six years. Today, Peter is senior C++ engineer and Scrum team leader at Medicim, the R&D unit for digital dentistry software of Envista Holdings. At Medicim, he codevelops a suite of applications for dental professionals, capable of capturing patient data from a wide range of hardware, with advanced diagnostic functionality and support for implant planning and prosthetic design.

Common themes in his professional career include advanced desktop application development, mastering and refactoring of code bases of millions of lines of C++ code, high-performant, real-time processing of 3-D data, concurrency, algorithms and data structures, interfacing with cutting-edge hardware, and leading agile development teams.

Peter is a regular speaker at, and board member of, the Belgian C++ Users Group. He also co-authored two books: *C++ Standard Library Quick Reference* and *Beginning C++* (5th edition), both published by Apress.

OCKERT J. DU PREEZ is a self-taught developer who started learning programming in the days of QBasic. He has written hundreds of developer articles over the years detailing his programming quests and adventures. His articles can be found on CodeGuru (codeguru.com), Developer.com (developer.com), DevX (devx.com), and Database Journal (databasejournal.com). Software development is his second love, just after his wife and child.

He knows a broad spectrum of development languages including C++, C#, VB.NET, JavaScript, and HTML. He has written the books *Visual Studio 2019 In-Depth* (BpB Publications) and *JavaScript for Gurus* (BpB Publications).

He was a Microsoft Most Valuable Professional for .NET (2008–2017).

ACKNOWLEDGMENTS

I THANK THE JOHN WILEY & SONS AND WROX PRESS editorial and production teams for their support. Especially, thank you to Jim Minatel, executive editor at Wiley, for giving me a chance to write this fifth edition; Kelly Talbot, project editor, for managing this project; and Kim Wimpsett, copy editor, for improving readability and consistency and making sure the text is grammatically correct.

Thanks to technical editor Hannes Du Preez for checking the technical accuracy of the book. His contributions in strengthening this book are greatly appreciated.

A very special thank you to technical editor Peter Van Weert for his outstanding contributions. His considerable advice and insights have truly elevated this book to a higher level.

Of course, the support and patience of my parents and my brother were very important in finishing this book. I would also like to express my sincere gratitude to my employer, Nikon Metrology, for supporting me during this project.

Finally, I thank you, the reader, for trying this approach to professional C++ software development.

—MARC GREGOIRE

CONTENTS

INTRODUCTION

xlvii

PART I: INTRODUCTION TO PROFESSIONAL C++

CHAPTER 1: A CRASH COURSE IN C++ AND THE STANDARD LIBRARY 3

C++ Crash Course	4
The Obligatory “Hello, World” Program	4
Comments	5
Importing Modules	5
Preprocessor Directives	5
The main() Function	8
I/O Streams	8
Namespaces	9
Nested Namespace	11
Namespace Alias	11
Literals	11
Variables	12
Numerical Limits	14
Zero Initialization	15
Casting	15
Floating-Point Numbers	16
Operators	16
Enumerated Types	19
Old-Style Enumerated Types	21
Structs	22
Conditional Statements	23
if/else Statements	23
switch Statements	24
The Conditional Operator	25
Logical Evaluation Operators	26
Three-Way Comparisons	27
Functions	28
Function Return Type Deduction	30
Current Function’s Name	30
Function Overloading	30

Attributes	30
[[nodiscard]]	31
[[maybe_unused]]	31
[[noreturn]]	32
[[deprecated]]	32
[[likely]] and [[unlikely]]	33
C-Style Arrays	33
std::array	35
std::vector	36
std::pair	36
std::optional	37
Structured Bindings	38
Loops	38
The while Loop	38
The do/while Loop	39
The for Loop	39
The Range-Based for Loop	39
Initializer Lists	40
Strings in C++	40
C++ as an Object-Oriented Language	41
Defining Classes	41
Using Classes	44
Scope Resolution	44
Uniform Initialization	45
Designated Initializers	48
Pointers and Dynamic Memory	49
The Stack and the Free Store	49
Working with Pointers	50
Dynamically Allocated Arrays	51
Null Pointer Constant	52
The Use of const	53
const as a Qualifier for a Type	53
const Methods	55
The constexpr Keyword	56
The consteval Keyword	57
References	58
Reference Variables	58
Reference Data Members	61
Reference Parameters	61
Reference Return Values	64
Deciding Between References and Pointers	64

const_cast()	68
Exceptions	69
Type Aliases	70
typedefs	71
Type Inference	72
The auto Keyword	72
The decltype Keyword	75
The Standard Library	75
Your First Bigger C++ Program	75
An Employee Records System	76
The Employee Class	76
Employee.cppm	76
Employee.cpp	78
EmployeeTest.cpp	79
The Database Class	80
Database.cppm	80
Database.cpp	81
DatabaseTest.cpp	82
The User Interface	82
Evaluating the Program	85
Summary	85
Exercises	85
CHAPTER 2: WORKING WITH STRINGS AND STRING VIEWS	87
Dynamic Strings	88
C-Style Strings	88
String Literals	90
Raw String Literals	90
The C++ std::string Class	92
What Is Wrong with C-Style Strings?	92
Using the string Class	92
std::string Literals	95
CTAD with std::vector and Strings	96
Numeric Conversions	96
High-Level Numeric Conversions	96
Low-Level Numeric Conversions	97
The std::string_view Class	100
std::string_view and Temporary Strings	102
std::string_view Literals	102
Nonstandard Strings	102
String Formatting	103
Format Specifiers	104

width	104
[fill]align	105
sign	105
#	105
type	106
precision	107
0	107
Format Specifier Errors	107
Support for Custom Types	107
Summary	110
Exercises	110
CHAPTER 3: CODING WITH STYLE	111
The Importance of Looking Good	111
Thinking Ahead	112
Elements of Good Style	112
Documenting Your Code	112
Reasons to Write Comments	112
Commenting to Explain Usage	112
Commenting to Explain Complicated Code	115
Commenting to Convey Meta-information	116
Commenting Styles	117
Commenting Every Line	117
Prefix Comments	118
Fixed-Format Comments	119
Ad Hoc Comments	120
Self-Documenting Code	122
Decomposition	122
Decomposition Through Refactoring	123
Decomposition by Design	124
Decomposition in This Book	124
Naming	124
Choosing a Good Name	124
Naming Conventions	125
Counters	125
Prefixes	126
Hungarian Notation	126
Getters and Setters	127
Capitalization	127
Namespaced Constants	127
Using Language Features with Style	127
Use Constants	128

Use References Instead of Pointers	128
Use Custom Exceptions	129
Formatting	129
The Curly Brace Alignment Debate	130
Coming to Blows over Spaces and Parentheses	131
Spaces, Tabs, and Line Breaks	131
Stylistic Challenges	132
Summary	132
Exercises	133

PART II: PROFESSIONAL C++ SOFTWARE DESIGN

CHAPTER 4: DESIGNING PROFESSIONAL C++ PROGRAMS	137
What Is Programming Design?	138
The Importance of Programming Design	139
Designing for C++	141
Two Rules for Your Own C++ Designs	142
Abstraction	142
Benefiting from Abstraction	142
Incorporating Abstraction in Your Design	143
Reuse	144
Writing Reusable Code	144
Reusing Designs	145
Reusing Existing Code	146
A Note on Terminology	146
Deciding Whether to Reuse Code or Write it Yourself	147
Advantages to Reusing Code	147
Disadvantages to Reusing Code	148
Putting It Together to Make a Decision	149
Guidelines for Choosing a Library to Reuse	149
Understand the Capabilities and Limitations	149
Understand the Learning Cost	150
Understand the Performance	150
Understand Platform Limitations	153
Understand Licensing	153
Understand Support and Know Where to Find Help	154
Prototype	154
Open-Source Libraries	155
The C++ Standard Library	157

Designing a Chess Program	157
Requirements	158
Design Steps	158
Divide the Program into Subsystems	158
Choose Threading Models	160
Specify Class Hierarchies for Each Subsystem	161
Specify Classes, Data Structures, Algorithms, and Patterns for Each Subsystem	162
Specify Error Handling for Each Subsystem	165
Summary	166
Exercises	166
CHAPTER 5: DESIGNING WITH OBJECTS	169
<hr/>	
Am I Thinking Procedurally?	170
The Object-Oriented Philosophy	170
Classes	170
Components	171
Properties	171
Behaviors	172
Bringing It All Together	172
Living in a World of Classes	173
Over-Classification	173
Overly General Classes	174
Class Relationships	175
The Has-a Relationship	175
The Is-a Relationship (Inheritance)	176
Inheritance Techniques	177
Polymorphism	178
The Fine Line Between Has-a and Is-a	178
The Not-a Relationship	181
Hierarchies	182
Multiple Inheritance	183
Mixin Classes	184
Summary	185
Exercises	185
CHAPTER 6: DESIGNING FOR REUSE	187
<hr/>	
The Reuse Philosophy	188
How to Design Reusable Code	189
Use Abstraction	189
Structure Your Code for Optimal Reuse	191
Avoid Combining Unrelated or Logically Separate Concepts	191

Use Templates for Generic Data Structures and Algorithms	193
Provide Appropriate Checks and Safeguards	195
Design for Extensibility	196
Design Usable Interfaces	198
Consider the Audience	198
Consider the Purpose	199
Design Interfaces That Are Easy to Use	200
Design General-Purpose Interfaces	204
Reconciling Generality and Ease of Use	205
Designing a Successful Abstraction	205
The SOLID Principles	206
Summary	207
Exercises	207

PART III: C++ CODING THE PROFESSIONAL WAY

CHAPTER 7: MEMORY MANAGEMENT	211
Working with Dynamic Memory	212
How to Picture Memory	212
Allocation and Deallocation	213
Using new and delete	213
What About My Good Friend malloc?	214
When Memory Allocation Fails	215
Arrays	215
Arrays of Primitive Types	215
Arrays of Objects	218
Deleting Arrays	218
Multidimensional Arrays	219
Working with Pointers	223
A Mental Model for Pointers	223
Casting with Pointers	224
Array-Pointer Duality	224
Arrays Are Pointers!	224
Not All Pointers Are Arrays!	226
Low-Level Memory Operations	227
Pointer Arithmetic	227
Custom Memory Management	228
Garbage Collection	228
Object Pools	229
Common Memory Pitfalls	229
Underallocating Data Buffers and Out-of-Bounds Memory Access	229

Memory Leaks	231
Finding and Fixing Memory Leaks in Windows with Visual C++	232
Finding and Fixing Memory Leaks in Linux with Valgrind	233
Double-Deletion and Invalid Pointers	234
Smart Pointers	234
unique_ptr	235
Creating unique_ptrs	236
Using unique_ptrs	237
unique_ptr and C-Style Arrays	238
Custom Deleters	239
shared_ptr	239
Creating and Using shared_ptrs	239
The Need for Reference Counting	241
Casting a shared_ptr	242
Aliasing	242
weak_ptr	243
Passing to Functions	244
Returning from Functions	244
enable_shared_from_this	244
The Old and Removed auto_ptr	245
Summary	246
Exercises	246

CHAPTER 8: GAINING PROFICIENCY WITH CLASSES AND OBJECTS **249**

Introducing the Spreadsheet Example	250
Writing Classes	250
Class Definitions	250
Class Members	251
Access Control	251
Order of Declarations	252
In-Class Member Initializers	253
Defining Methods	253
Accessing Data Members	254
Calling Other Methods	254
The this Pointer	255
Using Objects	257
Objects on the Stack	257
Objects on the Free Store	257
Understanding Object Life Cycles	258
Object Creation	258
Writing Constructors	259
Using Constructors	260

Providing Multiple Constructors	260
Default Constructors	261
Constructor Initializers	265
Copy Constructors	269
Initializer-List Constructors	271
Delegating Constructors	273
Converting Constructors and Explicit Constructors	273
Summary of Compiler-Generated Constructors	275
Object Destruction	276
Assigning to Objects	277
Declaring an Assignment Operator	278
Defining an Assignment Operator	278
Explicitly Defaulted and Deleted Assignment Operator	280
Compiler-Generated Copy Constructor and Copy Assignment Operator	280
Distinguishing Copying from Assignment	280
Objects as Return Values	280
Copy Constructors and Object Members	281
Summary	282
Exercises	282
CHAPTER 9: MASTERING CLASSES AND OBJECTS	283
Friends	284
Dynamic Memory Allocation in Objects	285
The Spreadsheet Class	285
Freeing Memory with Destructors	288
Handling Copying and Assignment	289
The Spreadsheet Copy Constructor	291
The Spreadsheet Assignment Operator	291
Disallowing Assignment and Pass-by-Value	294
Handling Moving with Move Semantics	295
Rvalue References	295
Implementing Move Semantics	297
Testing the Spreadsheet Move Operations	301
Implementing a Swap Function with Move Semantics	303
Using <code>std::move()</code> in Return Statements	303
Optimal Way to Pass Arguments to Functions	304
Rule of Zero	305
More About Methods	306
static Methods	306
const Methods	307
mutable Data Members	308

Method Overloading	308
Overloading Based on const	309
Explicitly Deleting Overloads	310
Ref-Qualified Methods	310
Inline Methods	311
Default Arguments	313
Different Kinds of Data Members	314
static Data Members	314
Inline Variables	314
Accessing static Data Members within Class Methods	315
Accessing static Data Members Outside Methods	316
const static Data Members	316
Reference Data Members	317
Nested Classes	318
Enumerated Types Inside Classes	319
Operator Overloading	320
Example: Implementing Addition for SpreadsheetCells	320
First Attempt: The add Method	320
Second Attempt: Overloaded operator+ as a Method	321
Third Attempt: Global operator+	322
Overloading Arithmetic Operators	324
Overloading the Arithmetic Shorthand Operators	324
Overloading Comparison Operators	325
Compiler-Generated Comparison Operators	328
Building Types with Operator Overloading	330
Building Stable Interfaces	330
Using Interface and Implementation Classes	330
Summary	334
Exercises	335
CHAPTER 10: DISCOVERING INHERITANCE TECHNIQUES	337
Building Classes with Inheritance	338
Extending Classes	338
A Client's View of Inheritance	339
A Derived Class's View of Inheritance	340
Preventing Inheritance	341
Overriding Methods	342
The virtual Keyword	342
Syntax for Overriding a Method	342
A Client's View of Overridden Methods	343
The override Keyword	344

The Truth About virtual	346
Preventing Overriding	350
Inheritance for Reuse	350
The WeatherPrediction Class	350
Adding Functionality in a Derived Class	351
Replacing Functionality in a Derived Class	352
Respect Your Parents	353
Parent Constructors	353
Parent Destructors	355
Referring to Parent Names	356
Casting Up and Down	358
Inheritance for Polymorphism	360
Return of the Spreadsheet	360
Designing the Polymorphic Spreadsheet Cell	360
The SpreadsheetCell Base Class	361
A First Attempt	361
Pure Virtual Methods and Abstract Base Classes	362
The Individual Derived Classes	363
StringSpreadsheetCell Class Definition	363
StringSpreadsheetCell Implementation	363
DoubleSpreadsheetCell Class Definition and Implementation	364
Leveraging Polymorphism	364
Future Considerations	365
Multiple Inheritance	367
Inheriting from Multiple Classes	367
Naming Collisions and Ambiguous Base Classes	368
Name Ambiguity	368
Ambiguous Base Classes	369
Uses for Multiple Inheritance	371
Interesting and Obscure Inheritance Issues	371
Changing the Overridden Method's Return Type	371
Adding Overloads of Virtual Base Class Methods to Derived Classes	373
Inherited Constructors	374
Hiding of Inherited Constructors	375
Inherited Constructors and Multiple Inheritance	376
Initialization of Data Members	377
Special Cases in Overriding Methods	378
The Base Class Method Is static	378
The Base Class Method Is Overloaded	379
The Base Class Method Is private	380
The Base Class Method Has Default Arguments	382

The Base Class Method Has a Different Access Specification	383
Copy Constructors and Assignment Operators in Derived Classes	385
Run-Time Type Facilities	386
Non-public Inheritance	388
Virtual Base Classes	389
Casts	390
static_cast()	390
reinterpret_cast()	391
std::bit_cast()	392
dynamic_cast()	393
Summary of Casts	394
Summary	394
Exercises	395
CHAPTER 11: ODDS AND ENDS	397
<hr/>	
Modules	397
Module Interface Files	399
Module Implementation Files	401
Splitting Interface from Implementation	402
Visibility vs. Reachability	403
Submodules	404
Module Partitions	405
Implementation Partitions	407
Header Units	408
Header Files	408
Duplicate Definitions	409
Circular Dependencies	409
Querying Existence of Headers	410
Feature Test Macros for Core Language Features	410
The static Keyword	411
static Data Members and Methods	411
static Linkage	411
The extern Keyword	413
static Variables in Functions	414
Order of Initialization of Nonlocal Variables	415
Order of Destruction of Nonlocal Variables	415

C Utilities	415
Variable-Length Argument Lists	415
Accessing the Arguments	416
Why You Shouldn't Use C-Style Variable-Length Argument Lists	417
Preprocessor Macros	417
Summary	419
Exercises	419
CHAPTER 12: WRITING GENERIC CODE WITH TEMPLATES	421
<hr/>	
Overview of Templates	422
Class Templates	422
Writing a Class Template	423
Coding Without Templates	423
A Template Grid Class	426
Using the Grid Template	430
How the Compiler Processes Templates	431
Selective Instantiation	431
Template Requirements on Types	432
Distributing Template Code Between Files	432
Method Definitions in Same File as Class Template Definition	433
Method Definitions in Separate File	433
Template Parameters	433
Non-type Template Parameters	434
Default Values for Type Parameters	436
Class Template Argument Deduction	436
Method Templates	438
Method Templates with Non-type Parameters	440
Class Template Specialization	442
Deriving from Class Templates	445
Inheritance vs. Specialization	446
Alias Templates	447
Function Templates	447
Function Template Overloading	449
Friend Function Templates of Class Templates	449
More on Template Parameter Deduction	451
Return Type of Function Templates	451
Abbreviated Function Template Syntax	453

Variable Templates	454
Concepts	454
Syntax	455
Constraints Expression	455
Requires Expressions	455
Combining Concept Expressions	457
Predefined Standard Concepts	457
Type-Constrained auto	458
Type Constraints and Function Templates	458
Constraint Subsumption	460
Type Constraints and Class Templates	461
Type Constraints and Class Methods	461
Type Constraints and Template Specialization	462
Summary	463
Exercises	463
CHAPTER 13: DEMYSTIFYING C++ I/O	465
Using Streams	466
What Is a Stream, Anyway?	466
Stream Sources and Destinations	467
Output with Streams	468
Output Basics	468
Methods of Output Streams	469
Handling Output Errors	470
Output Manipulators	471
Input with Streams	473
Input Basics	473
Handling Input Errors	475
Input Methods	476
Input Manipulators	480
Input and Output with Objects	481
Custom Manipulators	482
String Streams	482
File Streams	484
Text Mode vs. Binary Mode	485
Jumping Around with seek() and tell()	485
Linking Streams Together	487
Bidirectional I/O	488
Filesystem Support Library	490
Path	490
Directory Entry	491

Helper Functions	492
Directory Iteration	492
Summary	493
Exercises	493
CHAPTER 14: HANDLING ERRORS	495
<hr/>	
Errors and Exceptions	496
What Are Exceptions, Anyway?	496
Why Exceptions in C++ Are a Good Thing	496
Recommendation	498
Exception Mechanics	498
Throwing and Catching Exceptions	499
Exception Types	501
Catching Exception Objects as Reference-to-const	502
Throwing and Catching Multiple Exceptions	503
Matching and const	505
Matching Any Exception	505
Uncaught Exceptions	505
noexcept Specifier	507
noexcept(expression) Specifier	508
noexcept(expression) Operator	508
Throw Lists	508
Exceptions and Polymorphism	509
The Standard Exception Hierarchy	509
Catching Exceptions in a Class Hierarchy	510
Writing Your Own Exception Classes	512
Source Location	514
Nested Exceptions	517
Rethrowing Exceptions	519
Stack Unwinding and Cleanup	520
Use Smart Pointers	521
Catch, Cleanup, and Rethrow	522
Common Error-Handling Issues	523
Memory Allocation Errors	523
Non-throwing new	524
Customizing Memory Allocation Failure Behavior	524
Errors in Constructors	526
Function-Try-Blocks for Constructors	528
Errors in Destructors	531
Summary	531
Exercises	532

CHAPTER 15: OVERLOADING C++ OPERATORS	535
Overview of Operator Overloading	536
Why Overload Operators?	536
Limitations to Operator Overloading	536
Choices in Operator Overloading	537
Method or Global Function	537
Choosing Argument Types	538
Choosing Return Types	538
Choosing Behavior	539
Operators You Shouldn't Overload	539
Summary of Overloadable Operators	540
Rvalue References	544
Precedence and Associativity	545
Relational Operators	546
Overloading the Arithmetic Operators	547
Overloading Unary Minus and Unary Plus	547
Overloading Increment and Decrement	547
Overloading the Bitwise and Binary Logical Operators	548
Overloading the Insertion and Extraction Operators	549
Overloading the Subscripting Operator	550
Providing Read-Only Access with operator[]	553
Non-integral Array Indices	555
Overloading the Function Call Operator	555
Overloading the Dereferencing Operators	557
Implementing operator*	558
Implementing operator->	558
What in the World Are operator.* and operator->*?	559
Writing Conversion Operators	559
Operator auto	560
Solving Ambiguity Problems with Explicit Conversion Operators	561
Conversions for Boolean Expressions	561
Overloading the Memory Allocation and Deallocation Operators	563
How new and delete Really Work	564
The New-Expression and operator new	564
The Delete-Expression and operator delete	565
Overloading operator new and operator delete	565
Explicitly Deleting/Defaulting operator new and operator delete	568
Overloading operator new and operator delete with Extra Parameters	568
Overloading operator delete with Size of Memory as Parameter	569
Overloading User-Defined Literal Operators	570

Cooked-Mode Literal Operator	570
Raw-Mode Literal Operator	571
Standard User-Defined Literals	571
Summary	572
Exercises	572
CHAPTER 16: OVERVIEW OF THE C++ STANDARD LIBRARY	573
<hr/>	
Coding Principles	574
Use of Templates	574
Use of Operator Overloading	575
Overview of the C++ Standard Library	575
Strings	575
Regular Expressions	576
I/O Streams	576
Smart Pointers	576
Exceptions	576
Numerics Library	577
Time and Date Utilities	579
Random Numbers	579
Initializer Lists	579
Pair and Tuple	579
Vocabulary Types	580
Function Objects	580
Filesystem	580
Multithreading	580
Type Traits	581
Standard Integer Types	581
Standard Library Feature Test Macros	581
<version>	582
Source Location	582
Containers	582
vector	583
list	584
forward_list	584
deque	584
array	584
span	585
queue	585
priority_queue	585
stack	586

set and multiset	586
map and multimap	587
Unordered Associative Containers/Hash Tables	587
bitset	588
Summary of Standard Library Containers	588
Algorithms	591
Nonmodifying Sequence Algorithms	591
Modifying Sequence Algorithms	593
Operational Algorithms	595
Swap Algorithms	595
Partition Algorithms	595
Sorting Algorithms	596
Binary Search Algorithms	597
Set Algorithms	597
Heap Algorithms	598
Minimum/Maximum Algorithms	598
Numerical Processing Algorithms	599
Permutation Algorithms	600
Choosing an Algorithm	600
Ranges Library	601
What's Missing from the Standard Library	601
Summary	601
Exercises	601
CHAPTER 17: UNDERSTANDING ITERATORS AND THE RANGES LIBRARY	603
Iterators	604
Getting Iterators for Containers	606
Iterator Traits	608
Examples	609
Stream Iterators	610
Output Stream Iterator	610
Input Stream Iterator	611
Iterator Adapters	612
Insert Iterators	612
Reverse Iterators	614
Move Iterators	615
Ranges	616
Range-Based Algorithms	617
Projection	618
Views	619

Modifying Elements Through a View	622
Mapping Elements	623
Range Factories	623
Input Streams as Views	625
Summary	625
Exercises	626
CHAPTER 18: STANDARD LIBRARY CONTAINERS	627
<hr/>	
Containers Overview	628
Requirements on Elements	628
Exceptions and Error Checking	630
Sequential Containers	631
vector	631
vector Overview	631
vector Details	633
Move Semantics	646
vector Example: A Round-Robin Class	647
The vector<bool> Specialization	652
deque	653
list	653
Accessing Elements	653
Iterators	654
Adding and Removing Elements	654
list Size	654
Special list Operations	654
list Example: Determining Enrollment	656
forward_list	657
array	660
span	661
Container Adapters	663
queue	663
queue Operations	663
queue Example: A Network Packet Buffer	664
priority_queue	666
priority_queue Operations	666
priority_queue Example: An Error Correlator	667
stack	668
stack Operations	668
stack Example: Revised Error Correlator	669
Ordered Associative Containers	669
The pair Utility Class	669
map	670

Constructing maps	670
Inserting Elements	671
map Iterators	674
Looking Up Elements	675
Removing Elements	675
Nodes	676
map Example: Bank Account	676
multimap	679
multimap Example: Buddy Lists	680
set	682
set Example: Access Control List	682
multiset	684
Unordered Associative Containers or Hash Tables	684
Hash Functions	684
unordered_map	686
unordered_map Example: Phone Book	689
unordered_multimap	690
unordered_set/unordered_multiset	691
Other Containers	691
Standard C-Style Arrays	691
Strings	692
Streams	693
bitset	693
bitset Basics	693
Bitwise Operators	694
bitset Example: Representing Cable Channels	694
Summary	697
Exercises	698
CHAPTER 19: FUNCTION POINTERS, FUNCTION OBJECTS, AND LAMBDA EXPRESSIONS	699
<hr/>	
Function Pointers	700
Pointers to Methods (and Data Members)	702
std::function	703
Function Objects	705
Writing Your First Function Object	705
Function Objects in the Standard Library	706
Arithmetic Function Objects	706
Comparison Function Objects	707
Logical Function Objects	709
Bitwise Function Objects	709
Adapter Function Objects	709

Lambda Expressions	713
Syntax	713
Lambda Expressions as Parameters	718
Generic Lambda Expressions	719
Lambda Capture Expressions	719
Templated Lambda Expressions	720
Lambda Expressions as Return Type	721
Lambda Expressions in Unevaluated Contexts	722
Default Construction, Copying, and Assigning	722
Invokers	722
Summary	723
Exercises	723
CHAPTER 20: MASTERING STANDARD LIBRARY ALGORITHMS	725
<hr/>	
Overview of Algorithms	726
The find and find_if Algorithms	726
The accumulate Algorithm	729
Move Semantics with Algorithms	730
Algorithm Callbacks	730
Algorithm Details	731
Non-modifying Sequence Algorithms	731
Search Algorithms	731
Specialized Searchers	733
Comparison Algorithms	733
Counting Algorithms	736
Modifying Sequence Algorithms	737
generate	737
transform	738
copy	739
move	740
replace	742
erase	742
remove	743
unique	744
shuffle	745
sample	745
reverse	746
Shifting Elements	746
Operational Algorithms	747
for_each	747
for_each_n	749

Partition Algorithms	749
Sorting Algorithms	750
Binary Search Algorithms	751
Set Algorithms	752
Minimum/Maximum Algorithms	755
Parallel Algorithms	756
Constrained Algorithms	758
Numerical Processing Algorithms	758
iota	759
Reduce Algorithms	759
Scan Algorithms	760
Summary	761
Exercises	761

CHAPTER 21: STRING LOCALIZATION AND REGULAR EXPRESSIONS **763**

Localization	763
Wide Characters	764
Localizing String Literals	764
Non-Western Character Sets	765
Locales and Facets	767
Using Locales	767
Global Locale	769
Character Classification	769
Character Conversion	769
Using Facets	770
Conversions	771
Regular Expressions	772
ECMAScript Syntax	773
Anchors	773
Wildcards	773
Alternation	773
Grouping	774
Repetition	774
Precedence	775
Character Set Matches	775
Word Boundaries	777
Back References	778
Lookahead	778
Regular Expressions and Raw String Literals	778
Common Regular Expressions	779
The regex Library	779

regex_match()	781
regex_match() Example	781
regex_search()	783
regex_search() Example	784
regex_iterator	784
regex_iterator Example	785
regex_token_iterator	785
regex_token_iterator Examples	786
regex_replace()	788
regex_replace() Examples	789
Summary	790
Exercises	791
CHAPTER 22: DATE AND TIME UTILITIES	793
Compile-Time Rational Numbers	794
Duration	796
Clock	801
Time Point	802
Date	804
Time Zone	807
Summary	808
Exercises	808
CHAPTER 23: RANDOM NUMBER FACILITIES	809
C-Style Random Number Generation	810
Random Number Engines	811
Random Number Engine Adapters	813
Predefined Engines and Engine Adapters	813
Generating Random Numbers	814
Random Number Distributions	816
Summary	819
Exercises	819
CHAPTER 24: ADDITIONAL LIBRARY UTILITIES	821
Vocabulary Types	821
variant	821
any	823
Tuples	824
Decompose Tuples	826
Structured Bindings	827
tie	827

Concatenation	828
Comparisons	828
make_from_tuple	829
apply	829
Summary	829
Exercises	830

PART IV: MASTERING ADVANCED FEATURES OF C++

CHAPTER 25: CUSTOMIZING AND EXTENDING THE STANDARD LIBRARY **833**

Allocators	834
Extending the Standard Library	835
Why Extend the Standard Library?	835
Writing a Standard Library Algorithm	836
find_all()	836
Writing a Standard Library Container	837
A Basic Directed Graph	837
Making directed_graph a Standard Library Container	848
Adding Support for Allocators	866
Improving graph_node	871
Additional Standard Library-Like Functionality	872
Further Improvements	874
Other Container Types	874
Summary	875
Exercises	875

CHAPTER 26: ADVANCED TEMPLATES **877**

More About Template Parameters	878
More About Template Type Parameters	878
Introducing Template Template Parameters	880
More About Non-type Template Parameters	882
Class Template Partial Specialization	884
Emulating Function Partial Specialization with Overloading	888
Template Recursion	889
An N-Dimensional Grid: First Attempt	889
A Real N-Dimensional Grid	890
Variadic Templates	892
Type-Safe Variable-Length Argument Lists	893

Variable Number of Mixin Classes	895
Fold Expressions	896
Metaprogramming	898
Factorial at Compile Time	898
Loop Unrolling	899
Printing Tuples	900
constexpr if	902
Using a Compile-Time Integer Sequence with Folding	903
Type Traits	903
Using Type Categories	905
Using Type Relationships	907
Using the conditional Type Trait	907
Using enable_if	909
Using constexpr if to Simplify enable_if Constructs	910
Logical Operator Traits	912
Static Assertions	912
Metaprogramming Conclusion	913
Summary	913
Exercises	913
CHAPTER 27: MULTITHREADED PROGRAMMING WITH C++	915
Introduction	916
Race Conditions	918
Tearing	919
Deadlocks	919
False-Sharing	920
Threads	921
Thread with Function Pointer	921
Thread with Function Object	922
Thread with Lambda	924
Thread with Member Function	924
Thread Local Storage	924
Canceling Threads	925
Automatically Joining Threads	925
Retrieving Results from Threads	926
Copying and Rethrowing Exceptions	926
Atomic Operations Library	929
Atomic Operations	931
Atomic Smart Pointers	932
Atomic References	932
Using Atomic Types	933
Waiting on Atomic Variables	935

Mutual Exclusion	936
Mutex Classes	936
Spinlock	936
Non-timed Mutex Classes	937
Timed Mutex Classes	939
Locks	939
lock_guard	939
unique_lock	940
shared_lock	941
Acquiring Multiple Locks at Once	941
scoped_lock	942
std::call_once	942
Examples Using Mutual Exclusion Objects	943
Thread-Safe Writing to Streams	943
Using Timed Locks	945
Double-Checked Locking	946
Condition Variables	947
Spurious Wake-Ups	948
Using Condition Variables	949
Latches	950
Barriers	951
Semaphores	951
Futures	952
std::promise and std::future	953
std::packaged_task	954
std::async	955
Exception Handling	956
std::shared_future	956
Example: Multithreaded Logger Class	958
Thread Pools	962
Coroutines	963
Threading Design and Best Practices	965
Summary	966
Exercises	966

PART V: C++ SOFTWARE ENGINEERING

CHAPTER 28: MAXIMIZING SOFTWARE ENGINEERING METHODS	971
The Need for Process	972
Software Life Cycle Models	973
The Waterfall Model	973

Benefits of the Waterfall Model	974
Drawbacks of the Waterfall Model	974
Sashimi Model	975
Spiral-like Models	975
Benefits of a Spiral-like Model	976
Drawbacks of a Spiral-like Model	977
Agile	978
Software Engineering Methodologies	978
The Unified Process	979
The Rational Unified Process	980
RUP as a Product	980
RUP as a Process	980
RUP in Practice	981
Scrum	981
Roles	981
The Process	982
Benefits of Scrum	983
Drawbacks of Scrum	983
eXtreme Programming	984
XP in Theory	984
XP in Practice	988
Software Triage	988
Building Your Own Process and Methodology	989
Be Open to New Ideas	989
Bring New Ideas to the Table	989
Recognize What Works and What Doesn't Work	989
Don't Be a Renegade	989
Source Code Control	990
Summary	992
Exercises	992
CHAPTER 29: WRITING EFFICIENT C++	993
Overview of Performance and Efficiency	994
Two Approaches to Efficiency	994
Two Kinds of Programs	994
Is C++ an Inefficient Language?	994
Language-Level Efficiency	995
Handle Objects Efficiently	996
Pass-by-Value or Pass-by-Reference	996
Return-by-Value or Return-by-Reference	998
Catch Exceptions by Reference	998

Use Move Semantics	998
Avoid Creating Temporary Objects	998
Return-Value Optimization	999
Pre-allocate Memory	1000
Use Inline Methods and Functions	1001
Design-Level Efficiency	1001
Cache Where Necessary	1002
Use Object Pools	1003
An Object Pool Implementation	1003
Using the Object Pool	1006
Profiling	1008
Profiling Example with gprof	1009
First Design Attempt	1009
Profiling the First Design Attempt	1012
Second Design Attempt	1014
Profiling the Second Design Attempt	1015
Profiling Example with Visual C++ 2019	1016
Summary	1019
Exercises	1019
CHAPTER 30: BECOMING ADEPT AT TESTING	1021
<hr/>	
Quality Control	1022
Whose Responsibility Is Testing?	1022
The Life Cycle of a Bug	1022
Bug-Tracking Tools	1023
Unit Testing	1025
Approaches to Unit Testing	1026
The Unit Testing Process	1026
Define the Granularity of Your Tests	1027
Brainstorm the Individual Tests	1028
Create Sample Data and Results	1029
Write the Tests	1029
Run the Tests	1030
Unit Testing in Action	1031
Introducing the Microsoft Visual C++ Testing Framework	1031
Writing the First Test	1033
Building and Running Tests	1034
Negative Tests	1034
Adding the Real Tests	1035

Debugging Tests	1038
Basking in the Glorious Light of Unit Test Results	1038
Fuzz Testing	1039
Higher-Level Testing	1039
Integration Tests	1039
Sample Integration Tests	1039
Methods of Integration Testing	1040
System Tests	1041
Regression Tests	1041
Tips for Successful Testing	1042
Summary	1043
Exercises	1043
CHAPTER 31: CONQUERING DEBUGGING	1045
<hr/>	
The Fundamental Law of Debugging	1046
Bug Taxonomies	1046
Avoiding Bugs	1046
Planning for Bugs	1047
Error Logging	1047
Debug Traces	1049
Debug Mode	1049
Ring Buffers	1053
Assertions	1057
Crash Dumps	1058
Debugging Techniques	1059
Reproducing Bugs	1059
Debugging Reproducible Bugs	1060
Debugging Nonreproducible Bugs	1060
Debugging Regressions	1061
Debugging Memory Problems	1062
Categories of Memory Errors	1062
Tips for Debugging Memory Errors	1065
Debugging Multithreaded Programs	1066
Debugging Example: Article Citations	1067
Buggy Implementation of an ArticleCitations Class	1067
Testing the ArticleCitations Class	1070
Lessons from the ArticleCitations Example	1079
Summary	1079
Exercises	1080

CHAPTER 32: INCORPORATING DESIGN TECHNIQUES AND FRAMEWORKS	1083
"I Can Never Remember How to . . ."	1084
. . .Write a Class	1084
. . .Derive from an Existing Class	1086
. . .Write a Lambda Expression	1086
. . .Use the Copy-and-Swap Idiom	1087
. . .Throw and Catch Exceptions	1088
. . .Write to a File	1089
. . .Read from a File	1089
. . .Write a Class Template	1090
. . .Constrain Template Parameters	1090
There Must Be a Better Way	1091
Resource Acquisition Is Initialization	1091
Double Dispatch	1093
Attempt #1: Brute Force	1094
Attempt #2: Single Polymorphism with Overloading	1095
Attempt #3: Double Dispatch	1096
Mixin Classes	1098
Using Multiple Inheritance	1098
Using Class Templates	1100
Object-Oriented Frameworks	1101
Working with Frameworks	1101
The Model-View-Controller Paradigm	1102
Summary	1103
Exercises	1103
CHAPTER 33: APPLYING DESIGN PATTERNS	1105
Dependency Injection	1106
Example: A Logging Mechanism	1106
Implementation of a Dependency-Injected Logger	1106
Using Dependency Injection	1108
The Abstract Factory Pattern	1109
Example: A Car Factory Simulation	1109
Implementation of an Abstract Factory	1110
Using an Abstract Factory	1111
The Factory Method Pattern	1112
Example: A Second Car Factory Simulation	1112
Implementation of a Factory	1114
Using a Factory	1115

Other Types of Factories	1117
Other Uses of Factories	1117
The Adapter Pattern	1118
Example: Adapting a Logger Class	1118
Implementation of an Adapter	1119
Using an Adapter	1120
The Proxy Pattern	1120
Example: Hiding Network Connectivity Issues	1121
Implementation of a Proxy	1121
Using a Proxy	1122
The Iterator Pattern	1123
The Observer Pattern	1124
Example: Exposing Events from Subjects	1124
Implementation of an Observable	1124
Using an Observer	1125
The Decorator Pattern	1126
Example: Defining Styles in Web Pages	1127
Implementation of a Decorator	1127
Using a Decorator	1128
The Chain of Responsibility Pattern	1129
Example: Event Handling	1129
Implementation of a Chain of Responsibility	1129
Using a Chain of Responsibility	1131
The Singleton Pattern	1132
Example: A Logging Mechanism	1132
Implementation of a Singleton	1133
Using a Singleton	1135
Summary	1135
Exercises	1135
CHAPTER 34: DEVELOPING CROSS-PLATFORM AND CROSS-LANGUAGE APPLICATIONS	1137
<hr/>	
Cross-Platform Development	1138
Architecture Issues	1138
Size of Integers	1138
Binary Compatibility	1139
Address Sizes	1140
Byte Order	1140
Implementation Issues	1142
Compiler Quirks and Extensions	1142

Library Implementations	1142
Handling Different Implementations	1143
Platform-Specific Features	1143
Cross-Language Development	1145
Mixing C and C++	1145
Shifting Paradigms	1145
Linking with C Code	1149
Calling C++ Code from C#	1151
C++/CLI to Use C# Code from C++ and C++ from C#	1152
Calling C++ Code from Java with JNI	1154
Calling Scripts from C++ Code	1156
Calling C++ Code from Scripts	1156
A Practical Example: Encrypting Passwords	1157
Calling Assembly Code from C++	1159
Summary	1160
Exercises	1160

PART VI: APPENDICES

APPENDIX A: C++ INTERVIEWS	1165
APPENDIX B: ANNOTATED BIBLIOGRAPHY	1191
APPENDIX C: STANDARD LIBRARY HEADER FILES	1203
APPENDIX D: INTRODUCTION TO UML	1213
INDEX	1219

INTRODUCTION

The development of C++ started in 1982 by Bjarne Stroustrup, a Danish computer scientist, as the successor of C with Classes. In 1985, the first edition of *The C++ Programming Language* book was released. The first standardized version of C++ was released in 1998, called C++98. In 2003, C++03 came out and contained a few small updates. After that, it was silent for a while, but traction slowly started building up, resulting in a major update of the language in 2011, called C++11. From then on, the C++ Standard Committee has been on a three-year cycle to release updated versions, giving us C++14, C++17, and now C++20. All in all, with the release of C++20 in 2020, C++ is almost 40 years old and still going strong. In most rankings of programming languages in 2020, C++ is in the top four. It is being used on an extremely wide range of hardware, going from small devices with embedded microprocessors all the way up to multitrack supercomputers. Besides wide hardware support, C++ can be used to tackle almost any programming job, be it games on mobile platforms, performance-critical artificial intelligence (AI) and machine learning (ML) software, real-time 3-D graphics engines, low-level hardware drivers, entire operating systems, and so on. The performance of C++ programs is hard to match with any other programming language, and as such, it is the de facto language for writing fast, powerful, and enterprise-class object-oriented programs. As popular as C++ has become, the language is surprisingly difficult to grasp in full. There are simple, but powerful, techniques that professional C++ programmers use that don't show up in traditional texts, and there are useful parts of C++ that remain a mystery even to experienced C++ programmers.

Too often, programming books focus on the syntax of the language instead of its real-world use. The typical C++ text introduces a major part of the language in each chapter, explaining the syntax and providing an example. *Professional C++* does not follow this pattern. Instead of giving you just the nuts and bolts of the language with little practical context, this book will teach you how to use C++ in the real world. It will show you the little-known features that will make your life easier, as well as the programming techniques that separate novices from professional programmers.

WHO THIS BOOK IS FOR

Even if you have used the language for years, you might still be unfamiliar with the more advanced features of C++, or you might not be using the full capabilities of the language. Perhaps you write competent C++ code, but would like to learn more about design and good programming style in C++. Or maybe you're relatively new to C++ but want to learn the "right" way to program from the start. This book will meet those needs and bring your C++ skills to the professional level.

Because this book focuses on advancing from basic or intermediate knowledge of C++ to becoming a professional C++ programmer, it assumes that you have some knowledge about programming. Chapter 1, "A Crash Course in C++ and the Standard Library," covers the basics of C++ as a refresher, but it is not a substitute for actual training in programming. If you are just starting with C++ but you

have significant experience in another programming language such as C, Java, or C#, you should be able to pick up most of what you need from Chapter 1.

In any case, you should have a solid foundation in programming fundamentals. You should know about loops, functions, and variables. You should know how to structure a program, and you should be familiar with fundamental techniques such as recursion. You should have some knowledge of common data structures such as queues, and useful algorithms such as sorting and searching. You don't need to know about object-oriented programming just yet—that is covered in Chapter 5, “Designing with Objects.”

You will also need to be familiar with the compiler you will be using to compile your code. Two compilers, Microsoft Visual C++ and GCC, are introduced later in this introduction. For other compilers, refer to the documentation that came with your compiler.

WHAT THIS BOOK COVERS

Professional C++ uses an approach to C++ programming that will both increase the quality of your code and improve your programming efficiency. You will find discussions on new C++20 features throughout this fifth edition. These features are not just isolated to a few chapters or sections; instead, examples have been updated to use new features when appropriate.

Professional C++ teaches you more than just the syntax and language features of C++. It also emphasizes programming methodologies, reusable design patterns, and good programming style. The *Professional C++* methodology incorporates the entire software development process, from designing and writing code to debugging and working in groups. This approach will enable you to master the C++ language and its idiosyncrasies, as well as take advantage of its powerful capabilities for large-scale software development.

Imagine users who have learned all of the syntax of C++ without seeing a single example of its use. They know just enough to be dangerous! Without examples, they might assume that all code should go in the `main()` function of the program or that all variables should be global—practices that are generally not considered hallmarks of good programming.

Professional C++ programmers understand the correct way to use the language, in addition to the syntax. They recognize the importance of good design, the theories of object-oriented programming, and the best ways to use existing libraries. They have also developed an arsenal of useful code and reusable ideas.

By reading and understanding this book, you will become a professional C++ programmer. You will expand your knowledge of C++ to cover lesser known and often misunderstood language features. You will gain an appreciation for object-oriented design and acquire top-notch debugging skills. Perhaps most important, you will finish this book armed with a wealth of reusable ideas that you can actually apply to your daily work.

There are many good reasons to make the effort to be a professional C++ programmer as opposed to a programmer who knows C++. Understanding the true workings of the language will improve the quality of your code. Learning about different programming methodologies and processes will

help you to work better with your team. Discovering reusable libraries and common design patterns will improve your daily efficiency and help you stop reinventing the wheel. All of these lessons will make you a better programmer and a more valuable employee. While this book can't guarantee you a promotion, it certainly won't hurt.

HOW THIS BOOK IS STRUCTURED

This book is made up of five parts.

Part I, “Introduction to Professional C++,” begins with a crash course in C++ basics to ensure a foundation of C++ knowledge. Following the crash course, Part I goes deeper into working with strings, because strings are used extensively in most examples throughout the book. The last chapter of Part I explores how to write *readable* C++ code.

Part II, “Professional C++ Software Design,” discusses C++ design methodologies. You will read about the importance of design, the object-oriented methodology, and the importance of code reuse.

Part III, “C++ Coding the Professional Way,” provides a technical tour of C++ from the professional point of view. You will read about the best ways to manage memory in C++, how to create reusable classes, and how to leverage important language features such as inheritance. You will also learn techniques for input and output, error handling, string localization, how to work with regular expressions, and how to structure your code in reusable components called modules. You will read about how to implement operator overloading, how to write templates, how to put restrictions on template parameters using concepts, and how to unlock the power of lambda expressions and function objects. This part also explains the C++ Standard Library, including containers, iterators, ranges, and algorithms. You will also read about some additional libraries that are available in the standard, such as the libraries to work with time, dates, time zones, random numbers, and the filesystem.

Part IV, “Mastering Advanced Features of C++,” demonstrates how you can get the most out of C++. This part of the book exposes the mysteries of C++ and describes how to use some of its more advanced features. You will read about how to customize and extend the C++ Standard Library to your needs, advanced details on template programming, including template metaprogramming, and how to use multithreading to take advantage of multiprocessor and multicore systems.

Part V, “C++ Software Engineering,” focuses on writing enterprise-quality software. You'll read about the engineering practices being used by programming organizations today; how to write efficient C++ code; software testing concepts, such as unit testing and regression testing; techniques used to debug C++ programs; how to incorporate design techniques, frameworks, and conceptual object-oriented design patterns into your own code; and solutions for cross-language and cross-platform code.

The book concludes with a useful chapter-by-chapter guide to succeeding in a C++ technical interview, an annotated bibliography, a summary of the C++ header files available in the standard, and a brief introduction to the Unified Modeling Language (UML).

This book is not a reference of every single class, method, and function available in C++. The book *C++17 Standard Library Quick Reference* by Peter Van Weert and Marc Gregoire (Apress, 2019).

ISBN: 978-1-4842-4923-9) is a condensed reference to all essential data structures, algorithms, and functions provided by the C++ Standard Library up until the C++17 standard. Appendix B lists a couple more references. Two excellent online references are:

- cppreference.com: You can use this reference online or download an offline version for use when you are not connected to the Internet.
- cplusplus.com/reference/

When I refer to a “Standard Library Reference” in this book, I am referring to one of these detailed C++ references.

The following are additional excellent online resources:

- github.com/isocpp/CppCoreGuidelines: The *C++ Core Guidelines* are a collaborative effort led by Bjarne Stroustrup, inventor of the C++ language itself. They are the result of many person-years of discussion and design across a number of organizations. The aim of the guidelines is to help people to use modern C++ effectively. The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency.
- github.com/Microsoft/GSL: This is an implementation by Microsoft of the *Guidelines Support Library* (GSL) containing functions and types that are suggested for use by the C++ Core Guidelines. It’s a header-only library.
- isocpp.org/faq: This is a large collection of frequently asked C++ questions.
- stackoverflow.com: Search for answers to common programming questions, or ask your own questions.

CONVENTIONS

To help you get the most from the text and keep track of what’s happening, a number of conventions are used throughout this book.

WARNING *Boxes like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.*

NOTE *Tips, hints, tricks, and asides to the current discussion are placed in boxes like this one.*

As for styles in the text:

Important words are *italic* when they are introduced.

Keyboard strokes are shown like this: Ctrl+A.

Filenames and code within the text are shown like so: `monkey.cpp`.

URLs are shown like this: `wrox.com`.

Code is presented in three different ways:

```
// Comments in code are shown like this.
```

```
In code examples, new and important code is highlighted like this.
```

```
Code that's less important in the present context or that has been shown before is formatted like this.
```



Paragraphs or sections that are specific to the C++20 standard have a little C++20 icon on the left, just as this paragraph does. C++11, C++14, and C++17 features are not marked with any icon.

WHAT YOU NEED TO USE THIS BOOK

All you need to use this book is a computer with a C++ compiler. This book focuses only on parts of C++ that have been standardized, and not on vendor-specific compiler extensions.

Any C++ Compiler

You can use whichever C++ compiler you like. If you don't have a C++ compiler yet, you can download one for free. There are a lot of choices. For example, for Windows, you can download Microsoft Visual Studio Community Edition, which is free and includes Visual C++. For Linux, you can use GCC or Clang, which are also free.

The following two sections briefly explain how to use Visual C++ and GCC. Refer to the documentation that came with your compiler for more details.

COMPILERS AND C++20 FEATURE SUPPORT

This book discusses new features introduced with the C++20 standard. At the time of this writing, no compilers were fully C++20 compliant yet. Some new features were only supported by some compilers and not others, while other features were not yet supported by any compiler. Compiler vendors are hard at work to catch up with all new features, and I'm sure it won't take long before there will be fully C++20-compliant compilers available. You can keep track of which compiler supports which features at en.cppreference.com/w/cpp/compiler_support.

COMPILERS AND C++20 MODULE SUPPORT

At the time of this writing, there was no compiler available yet that fully supported C++20 modules. There was experimental support in some of the compilers, but it was still incomplete. This book uses modules everywhere. We did our best to make sure all sample code would compile once compilers fully support modules, but since we were not able to compile and test all examples, some errors might have crept in. When you use a compiler with support for modules and you encounter problems with any of the code samples, double-check the list of errata for the book at www.wiley.com/go/proc++5e to see if it's a known issue. If your compiler does not yet support modules, you can convert modularized code to non-modularized code, as explained briefly in Chapter 11, "Odds and Ends."

Example: Microsoft Visual C++ 2019

First, you need to create a project. Start Visual C++ 2019, and on the welcome screen, click the Create A New Project button. If the welcome screen is not shown, select File ⇨ New ⇨ Project. In the Create A New Project dialog, search for the Console App project template with tags C++, Windows, and Console, and click Next. Specify a name for the project and a location where to save it, and click Create.

Once your new project is loaded, you can see a list of project files in the Solution Explorer. If this docking window is not visible, select View ⇨ Solution Explorer. A newly created project will contain a file called `<projectname>.cpp`. You can start writing your C++ code in that `.cpp` file, or if you want to compile source code files from the downloadable source archive for this book, select the `<projectname>.cpp` file in the Solution Explorer and delete it. You can add new files or existing files to a project by right-clicking the project name in the Solution Explorer and then selecting Add ⇨ New Item or Add ⇨ Existing Item.

At the time of this writing, Visual C++ 2019 did not yet automatically enable C++20 features. To enable C++20 features, in the Solution Explorer window, right-click your project and click Properties. In the Properties window, go to Configuration Properties ⇨ C/C++ ⇨ Language, and set the C++ Language Standard option to ISO C++20 Standard or Preview - Features from the Latest C++ Working Draft, whichever is available in your version of Visual C++. These options are accessible only if your project contains at least one `.cpp` file.

Finally, select Build ⇨ Build Solution to compile your code. When it compiles without errors, you can run it with Debug ⇨ Start Debugging.

Module Support

At the time of this writing, Visual C++ 2019 did not yet have full support for modules. Authoring and consuming your own modules usually works just fine, but importing Standard Library headers such as the following did not yet work out of the box:

```
import <iostream>;
```

To make such import declarations work, for the time being you need to add a separate header file to your project, for example called `HeaderUnits.h`, which contains an import declaration for every Standard Library header you want to import. Here's an example:

```
// HeaderUnits.h
#pragma once
import <iostream>;
import <vector>;
import <optional>;
import <utility>;
// ...
```

Next, right-click the `HeaderUnits.h` file in the Solution Explorer and click Properties. In Configuration Properties ⇨ General, set Item Type to C/C++ Compiler and click Apply. Next, in Configuration Properties ⇨ C/C++ ⇨ Advanced, set Compile As to Compile as C++ Header Unit (/exportHeader) and click OK.

When you now recompile your project, all import declarations that have a corresponding import declaration in your `HeaderUnits.h` file should compile fine.

If you are using module implementation partitions (see Chapter 11), also known as internal partitions, then right-click all files containing such implementation partitions, click Properties, go to Configuration Properties ⇨ C/C++ ⇨ Advanced, and set the Compile As option to Compile as C++ Module Internal Partition (/internalPartition) and click OK.

Example: GCC

Create your source code files with any text editor you prefer and save them to a directory. To compile your code, open a terminal and run the following command, specifying all your `.cpp` files that you want to compile:

```
g++ -std=c++2a -o <executable_name> <source1.cpp> [source2.cpp ...]
```

The `-std=c++2a` option is required to tell GCC to enable C++20 support. This option will change to `-std=c++20` once GCC is fully C++20 compliant.

Module Support

At the time of this writing, GCC only had experimental support for modules through a special version of GCC (branch `devel/c++-modules`). When you are using such a version of GCC, module support is enabled with the `-fmodules-ts` option, which might change to `-fmodules` in the future.

Unfortunately, import declarations of Standard Library headers such as the following were not yet properly supported:

```
import <iostream>;
```

If that's the case, simply replace such import declarations with corresponding `#include` directives:

```
#include <iostream>
```

For example, the `AirlineTicket` example from Chapter 1 uses modules. After having replaced the imports for Standard Library headers with `#include` directives, you can compile the `AirlineTicket` example by changing to the directory containing the code and running the following command:

```
g++ -std=c++2a -fmodules-ts -o AirlineTicket AirlineTicket.cppm AirlineTicket.cpp
AirlineTicketTest.cpp
```

When it compiles without errors, you can run it as follows:

```
./AirlineTicket
```

std::format Support

Many code samples in this book use `std::format()`, introduced in Chapter 1. At the time of this writing, there was no compiler yet that had support for `std::format()`. However, as long as your compiler doesn't support `std::format()` yet, you can use the freely available `{fmt}` library as a drop-in replacement:

1. Download the latest version of the `{fmt}` library from <https://fmt.dev/> and extract the code on your machine.
2. Copy the `include/fmt` and `src` directories to `fmt` and `src` subdirectories in your project directory, and then add `fmt/core.h`, `fmt/format.h`, `fmt/format-inl.h`, and `src/format.cc` to your project.
3. Add a file called `format` (no extension) to the root directory of your project and add the following code to it:

```
#pragma once
#define FMT_HEADER_ONLY
#include "fmt/format.h"
namespace std
{
    using fmt::format;
    using fmt::format_error;
    using fmt::formatter;
}
```

4. Finally, add your project root directory (the directory containing the `format` file) as an additional include directory for your project. For example, in Visual C++, right click your project in the Solution Explorer, click Properties, go to Configuration Properties ⇄ C/C++ ⇄ General, and add `$(ProjectDir);` to the front of the Additional Include Directories option.

NOTE *Don't forget to undo these steps once your compiler supports the standard `std::format()`.*

READER SUPPORT FOR THIS BOOK

The following sections describe different options to get support for this book.

Companion Download Files

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. However, I suggest you type in all the code manually because it greatly benefits the learning process and your memory. All of the source code used in this book is available for download at www.wiley.com/go/proc++5e.

NOTE *Because many books have similar titles, you may find it easiest to search by ISBN; for this book, the ISBN is 978-1-119-69540-0.*

Once you've downloaded the code, just decompress it with your favorite decompression tool.

How to Contact the Publisher

If you believe you've found a mistake in this book, please bring it to our attention. At John Wiley & Sons, we understand how important it is to provide our customers with accurate content, but even with our best efforts an error may occur.

To submit your possible errata, please e-mail it to our Customer Service Team at wileysupport@wiley.com with "Possible Book Errata Submission" as a subject line.

How to Contact the Author

If you have any questions while reading this book, the author can easily be reached at marc.gregoire@nuonsoft.com and will try to get back to you in a timely manner.

PART I

Introduction to Professional C++

- ▶ CHAPTER 1: A Crash Course in C++ and the Standard Library
- ▶ CHAPTER 2: Working with Strings and String Views
- ▶ CHAPTER 3: Coding with Style



1

A Crash Course in C++ and the Standard Library

WHAT'S IN THIS CHAPTER?

- A brief overview of the most important parts and syntax of the C++ language and the Standard Library
- How to write a basic class
- How scope resolution works
- What uniform initialization is
- The use of `const`
- What pointers, references, exceptions, and type aliases are
- Basics of type inference

WILEY.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of the chapter's code download on this book's website at www.wiley.com/go/proc++5e on the Download Code tab.

The goal of this chapter is to cover briefly the most important parts of C++ so that you have a foundation of knowledge before embarking on the rest of this book. This chapter is not a comprehensive lesson in the C++ programming language or the Standard Library. Certain basic points, such as what a program is and what recursion is, are not covered. Esoteric points, such as the definition of a `union`, or the `volatile` keyword, are also omitted. Certain parts of the C language that are less relevant in C++ are also left out, as are parts of C++ that get in-depth coverage in later chapters.

This chapter aims to cover the parts of C++ that programmers encounter every day. For example, if you're fairly new to C++ and don't understand what a reference variable is, you'll learn about that kind of variable here. You'll also learn the basics of how to use the functionality available in the Standard Library, such as `vector` containers, optional values, `string` objects, and more. These parts of the Standard Library are briefly introduced in Chapter 1 so that these modern constructs can be used throughout examples in this book from the beginning.

If you already have significant experience with C++, skim this chapter to make sure that there aren't any fundamental parts of the language on which you need to brush up. If you're new to C++, read this chapter carefully and make sure you understand the examples. If you need additional introductory information, consult the titles listed in Appendix B.

C++ CRASH COURSE

The C++ language is often viewed as a “better C” or a “superset of C.” It was mainly designed to be an object-oriented C, commonly called as “C with classes.” Later on, many of the annoyances and rough edges of the C language were addressed as well. Because C++ is based on C, some of the syntax you'll see in this section will look familiar to you if you are an experienced C programmer. The two languages certainly have their differences, though. As evidence, *The C++ Programming Language* by C++ creator Bjarne Stroustrup (fourth edition; Addison-Wesley Professional, 2013) weighs in at 1,368 pages, while Kernighan and Ritchie's *The C Programming Language* (second edition; Prentice Hall, 1988) is a scant 274 pages. So, if you're a C programmer, be on the lookout for new or unfamiliar syntax!

The Obligatory “Hello, World” Program

In all its glory, the following code is the simplest C++ program you're likely to encounter:

```
// helloworld.cpp
import <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

This code, as you might expect, prints the message “Hello, World!” on the screen. It is a simple program and unlikely to win any awards, but it does exhibit the following important concepts about the format of a C++ program:

- Comments
- Importing modules
- The `main()` function
- I/O streams

These concepts are briefly explained in the following sections (along with header files as an alternative for modules, in the event that your compiler does not support C++20 modules yet).

Comments

The first line of the program is a *comment*, a message that exists for the programmer only and is ignored by the compiler. In C++, there are two ways to delineate a comment. In the preceding and following examples, two slashes indicate that whatever follows on that line is a comment:

```
// helloworld.cpp
```

The same behavior (this is to say, none) would be achieved by using a *multiline comment*. Multiline comments start with `/*` and end with `*/`. The following code shows a multiline comment in action (or, more appropriately, inaction):

```
/* This is a multiline comment.
   The compiler will ignore it.
  */
```

Comments are covered in detail in Chapter 3, “Coding with Style.”

C++20

Importing Modules

One of the bigger new features of C++20 is support for *modules*, replacing the old mechanism of so-called *header files*. If you want to use functionality from a module, you need to import that module. This is done with an `import` declaration. The first line of the “Hello, World” application imports the module called `<iostream>`, which declares the input and output mechanisms provided by C++:

```
import <iostream>;
```

If the program did not import that module, it would be unable to perform its only task of outputting text.

Since this is a book about C++20, this book uses modules everywhere. All functionality provided by the C++ Standard Library is provided in well-defined modules. Your own custom types and functionality can also be provided through self-written modules, as you will learn throughout this book. If your compiler does not yet support modules, simply replace `import` declarations with the proper `#include` preprocessor directives, discussed in the next section.

Preprocessor Directives

If your compiler does not yet support C++20 modules, then instead of an `import` declaration such as `import <iostream>;`, you need to write the following preprocessor directive:

```
#include <iostream>
```

In short, building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes meta-information about the code. Next, the code is *compiled*, or translated into machine-readable object files. Finally, the individual object files are *linked* together into a single application.

Directives aimed at the preprocessor start with the # character, as in the line `#include <iostream>` in the previous example. In this case, an `#include` directive tells the preprocessor to take everything from the `<iostream>` header file and make it available to the current file. The `<iostream>` header declares the input and output mechanisms provided by C++.

The most common use of header files is to declare functions that will be defined elsewhere. A function *declaration* tells the compiler how a function is called, declaring the number and types of parameters, and the function return type. A *definition* contains the actual code for the function. Before the introduction of modules in C++20, declarations usually went into *header files*, typically with extension `.h`, while definitions usually went into *source files*, typically with extension `.cpp`. With modules, it is no longer necessary to split declarations from definitions, although, as you will see, it is still possible to do so.

NOTE *In C, the names of the Standard Library header files usually end in `.h`, such as `<stdio.h>`, and namespaces are not used.*

In C++, the `.h` suffix is omitted for Standard Library headers, such as `<iostream>`, and everything is defined in the `std` namespace or a subnamespace of `std`.

The Standard Library headers from C still exist in C++ but in two versions.

- *The recommended versions without a `.h` suffix but with a `c` prefix. These versions put everything in the `std` namespace (for example, `<cstdio>`).*
- *The old versions with the `.h` suffix. These versions do not use namespaces (for example, `<stdio.h>`).*

Note that these C Standard Library headers are not guaranteed to be importable with an `import` declaration. To be safe, use `#include <xyz>` instead of `import <xyz>;`.

The following table shows some of the most common preprocessor directives:

PREPROCESSOR DIRECTIVE	FUNCTIONALITY	COMMON USES
<code>#include [file]</code>	The specified file is inserted into the code at the location of the directive.	Almost always used to include header files so that code can make use of functionality defined elsewhere.

PREPROCESSOR DIRECTIVE	FUNCTIONALITY	COMMON USES
<code>#define [id] [value]</code>	Every occurrence of the specified identifier is replaced with the specified value.	Often used in C to define a constant value or a macro. C++ provides better mechanisms for constants and most types of macros. Macros can be dangerous, so use them cautiously. See Chapter 11, "Odds and Ends," for details.
<code>#ifndef [id]</code> <code>#endif</code> <code>#ifndef [id]</code> <code>#endif</code>	Code within the <code>ifndef</code> ("if defined") or <code>ifndef</code> ("if not defined") blocks are conditionally included or omitted based on whether the specified identifier has been defined with <code>#define</code> .	Used most frequently to protect against circular includes. Each header file starts with an <code>ifndef</code> checking the absence of an identifier, followed by a <code>define</code> directive to define that identifier. The header file ends with an <code>endif</code> . This prevents the file from being included multiple times; see the example after this table.
<code>#pragma [xyz]</code>	<code>xyz</code> is compiler dependent. Most compilers support a <code>#pragma</code> to display a warning or error if the directive is reached during preprocessing.	See the example after this table.

One example of using preprocessor directives is to avoid multiple includes, as shown here:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif
```

If your compiler supports the `#pragma once` directive, and most modern compilers do, then this can be rewritten as follows:

```
#pragma once
// ... the contents of this header file
```

Chapter 11 discusses this in a bit more detail. But, as mentioned, this book uses C++20 modules instead of old-style header files.

The main() Function

`main()` is, of course, where the program starts. The return type of `main()` is an `int`, indicating the result status of the program. You can omit any explicit return statements in `main()`, in which case zero is returned automatically. The `main()` function either takes no parameters or takes two parameters as follows:

```
int main(int argc, char* argv[])
```

`argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that `argv[0]` can be the program name, but it might as well be an empty string, so do not rely on it; instead, use platform-specific functionality to retrieve the program name. The important thing to remember is that the actual arguments start at index 1.

I/O Streams

I/O streams are covered in depth in Chapter 13, “Demystifying C++ I/O,” but the basics of output and input are simple. Think of an output stream as a laundry chute for data. Anything you toss into it will be output appropriately. `std::cout` is the chute corresponding to the user console, or *standard out*. There are other chutes, including `std::cerr`, which outputs to the error console. The `<<` operator tosses data down the chute. In the preceding example, a quoted string of text is sent to standard out. Output streams allow multiple types of data to be sent down the stream sequentially on a single line of code. The following code outputs text, followed by a number, followed by more text:

```
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

Starting with C++20, though, it is recommended to use `std::format()`, defined in `<format>`, to perform string formatting. The `format()` function is discussed in detail in Chapter 2, “Working with Strings and String Views,” but in its most basic form it can be used to rewrite the previous statement as follows:

```
std::cout << std::format("There are {} ways I love you.", 219) << std::endl;
```

`std::endl` represents an end-of-line sequence. When the output stream encounters `std::endl`, it will output everything that has been sent down the chute so far and move to the next line. An alternate way of representing the end of a line is by using the `\n` character. The `\n` character is an *escape sequence*, which refers to a new-line character. Escape sequences can be used within any quoted string of text. The following table shows the most common ones:

ESCAPE SEQUENCE	MEANING
<code>\n</code>	New line: moves the cursor to the beginning of the next line
<code>\r</code>	Carriage return: moves the cursor to the beginning of the current line, but does not advance to the next line
<code>\t</code>	Tab
<code>\\</code>	Backslash character
<code>\"</code>	Quotation mark

WARNING *Keep in mind that `endl` inserts a new line into the stream and flushes everything currently in its buffers down the chute. Overusing `endl`, for example in a loop, is not recommended because it will have a performance impact. On the other hand, inserting `\n` into the stream also inserts a new line but does not automatically flush the buffers.*

Streams can also be used to accept input from the user. The simplest way to do this is to use the `>>` operator with an input stream. The `std::cin` input stream accepts keyboard input from the user. Here is an example:

```
int value;
std::cin >> value;
```

User input can be tricky because you can never know what kind of data the user will enter. See Chapter 13 for a full explanation of how to use input streams.

If you're new to C++ and coming from a C background, you're probably wondering what has been done with the trusty old `printf()` and `scanf()` functions. While these functions can still be used in C++, I strongly recommend using `format()` and the streams library instead, mainly because the `printf()` and `scanf()` family of functions do not provide any type safety.

Namespaces

Namespaces address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code. You can't change the library's function name, and it would be a big pain to change your own.

Namespaces come to the rescue in such scenarios because you can define the context in which names are defined. To place code in a namespace, enclose it within a namespace block. Here's an example:

```
namespace mycode {
    void foo()
    {
        std::cout << "foo() called in the mycode namespace" << std::endl;
    }
}
```

By placing your version of `foo()` in the namespace `mycode`, you are isolating it from the `foo()` function provided by the third-party library. To call the namespace-enabled version of `foo()`, prepend the namespace onto the function name by using `::`, also called the *scope resolution operator*, as follows:

```
mycode::foo(); // Calls the "foo" function in the "mycode" namespace
```

Any code that falls within a `mycode` namespace block can call other code within the same namespace without explicitly prepending the namespace. This implicit namespace is useful in making the code more readable. You can also avoid prepending of namespaces with a *using directive*. This directive